HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Robert Morgenstern

Untersuchung der Möglichkeiten der automatisierten Erstellung von Compilern anhand von Sprachdefinitionen zur Transformation von Quellcode aktueller Programmiersprachen untereinander

Mittweida, 2014

Fakultät Mathematik/Naturwissenschaften/Informatik

BACHELORARBEIT

Untersuchung der Möglichkeiten der automatisierten Erstellung von Compilern anhand von Sprachdefinitionen zur Transformation von Quellcode aktueller Programmiersprachen untereinander

Autor:
Herr

Robert Morgenstern

Studiengang:
Informatik

Seminargruppe:
IF10w1-B

Erstprüfer:
Prof. Dr.-Ing Jürgen Ruck

Zweitprüfer:
Dipl.-Päd. Ralf Neubert

Einreichung:
Mittweida, 25.05.2014

Verteidigung/Bewertung:
Mittweida, 2014

Faculty Math/Science/Computer-Science

BACHELORTHESIS

Exploration the possibilities of automated generation of compilers based on language definitions for the transformation of source code of actual programming languages

author:
Herr

Robert Morgenstern

course of studies:
Informatik

seminar group:
IF10w1-B

first examiner:
Prof. Dr.-Ing Jürgen Ruck

second examiner:
Dipl.-Päd. Ralf Neubert

submission:
Mittweida, 25.05.2014

defence/evaluation:
Mittweida, 2014

Bibliografische Beschreibung:

Morgenstern Robert:

Untersuchung der Möglichkeiten der automatisierten Erstellung von Compilern anhand von Sprachdefinitionen zur Transformation von Quellcode aktueller Programmiersprachen untereinander

50 Seiten Inhalt, 54 Abbildungen, 17 Seiten Anhang

Hochschule Mittweida, Fakultät Mathematik/Naturwissenschaften/Informatik,

Bachelorarbeit, 2014

Referat:

Das Ziel der Bachelorarbeit ist die Untersuchung der Möglichkeiten der automatisierten Erstellung eines Compilers. Dieser Compiler soll die Transformation von Quellcode einer Sprache in die einer anderen durchführen können.
Nach einem theoretischen Einstieg in das Thema Compilerbau wird die Turing eXtender Language, auf die Möglichkeit der Transformation von Quellcode, untersucht. Im Anschluss wird anhand eines Beispiels erklärt, wie eine Transformation mit der Turing eXtender Language möglich ist. Den Abschluss bildet eine Zusammenfassung über diese Arbeit.

Danksagung:

An dieser Stelle möchte ich mich bei allen Personen bedanken, welche mir bei der Erstellung dieser Arbeit unterstützend beigestanden haben.
Bedanken möchte ich mich bei meinem Hochschulbetreuer Herrn Professor Jürgen Ruck und bei meinem betrieblichen Betreuer Herrn Dipl.-Päd. Ralf Neubert, die es mir ermöglicht haben an diesem interessanten Thema zu arbeiten, mir entsprechend Freiheit gelassen haben das Thema nach eigenen Vorstellungen zu erarbeiten und mir immer hilfreiche Tipps und Motivation für das Thema gaben.
Ein großer Dank gebührt auch meiner Freundin Claudia, die mir während der gesamten Studienzeit und vor allem in der Zeit der Erarbeitung dieser Bachelorarbeit immer den Rücken freihielt und somit nicht zuletzt unserem gemeinsamen Sohn Jason nie das Gefühl vermittelte, einer stressigen Familiesituation ausgesetzt zu sein. Dafür mein ganz besonderer Dank.

Inhalt

Abbildungsverzeichnis

1 Einleitung

Bis 1949 bestanden Computerprogramme aus einer Aneinanderreihung von Nullen und Einsen. Dies ist die Sprache, die eine Maschine versteht und ausführen kann. Programmiersprachen der heutigen Zeit, wie beispielsweise Java oder C#, bestehen aus einer Kombination von Zeichen, die sich näher an der menschlichen Sprache orientiert als an der einer maschinellen. Diese Programme können von einer Maschine nur gelesen und zur Ausführung gebracht werden, weil es Programme gibt, die in der Lage sind, diese Kombinationen in eine für die Maschine ausführbare Form zu übersetzen. Diese Programme nennt man auch Compiler. Der erste Compiler wurde im Jahr 1949 von Grace Hopper entwickelt und nannte sich A-0. Der Bau eines Compilers ist eine Disziplin der Informatik und beschäftigt sich mit der Entwicklung und Programmierung solcher Programme. Mittlerweile existieren viele Programmiersprachen und Erweiterungen, welche alle kompiliert oder übersetzt werden müssen. Die Entwicklung solcher ist ein schnell voranschreitender, unaufhaltbarer Prozess der Zeit. Immer wieder werden neue Sprachen entwickelt oder weiterentwickelt. Schnell kann es dabei zu veralteten Programmen (engl. legacy system) kommen, welche nicht mehr effektiv genug arbeiten oder zu kostenintensiv betrieben werden müssen. Die Ablösung solcher Programme würde bedeuten, dass sie in einer aktuelleren Sprache neu programmiert oder gänzlich neu entwickelt werden müssen.

1.1 Zielsetzung

Ziel der vorliegenden Bachelorarbeit ist die Untersuchung der Möglichkeiten einen Compiler automatisiert zu erstellen, welcher in der Lage ist Quellcode einer Programmiersprache in den einer anderen zu transformieren.

Das Resultat der Arbeit soll zeigen, wie Transformationen möglich sind und wie sie durchgeführt werden.

1.2 Struktur der Arbeit

Kapitel 2 erklärt die theoretischen Grundlagen zum Thema Compilerbau. Es beschreibt dabei den Aufbau und die Funktion eines solchen, in welche Phasen er sich unterteilen lässt und andere Inhalte, welche zum besseren Verständnis der darauf folgenden Kapitel benötigt werden.

Kapitel 3 stellt Möglichkeiten vor, die es erlauben einzelne Bestandteile eines Compilers oder einen vollständigen Transcompiler automatisiert zu erstellen.

Kapitel 4 beschreibt und erklärt die Turing eXtender Language aus dem dritten Kapitel näher.

Kapitel 5 erklärt die Grundlagen der Turing eXtender Language.

Kapitel 6 beschreibt, wie es mit den im fünften Kapitel erklärten Grundlagen der Turing eXtender Language möglich ist eine Transformation, anhand eines Beispiels, durchzuführen.

Kapitel 7 bildet den Schluss dieser Arbeit. Es besteht aus einer Zusammenfassung, welche die gesamte Arbeit zusammenfassend auswertet und einem Ausblick.

2 Theoretische Grundlagen

2.1 Compiler

Ein Compiler (auch *Übersetzer*) ist ein Programm, das ein Quellprogramm einer höheren Programmiersprache, beispielsweise Java oder C#, in eine für den Computer ausführbare Form, den Maschinencode oder Assemblercode, übersetzt. Dieser erzeugte Code, in Form einer ausführbaren Datei, kann von einer Maschine gelesen und zur Ausführung gebracht werden. Den Prozess der Übersetzung nennt man auch Kompilierung oder Umwandlung.

Grafik: Prinzip eines Compilers

Abbildung 1: Prinzip eines Compilers

Übersetzt ein Compiler das Quellprogramm einer Programmiersprache nicht in Maschinencode sondern in Quellcode einer anderen Sprache so nennt man diesen Compiler auch Transcompiler oder kurz Transpiler (vgl. [WC14]).

2.2 Aufbau eines Compilers

In Abbildung 2 wird der schematischen Aufbau eines Compilers dargestellt. Sie zeigt, dass ein Compiler aus mehreren, logisch aufeinanderfolgenden Komponenten besteht, die sich wiederum in bestimmte Phasen aufteilen lassen.

Die **Analyse-Phase** wertet das Quellprogramm lexikalisch (**lexikalische Analyse**), syntaktisch (**syntaktische Analyse**) und semantisch aus. Sie speichert die Zwischenergebnisse der Auswertungen, zur weiteren Verarbeitung, in Symboltabellen und einem Syntaxbaum, beziehungsweise einem attributierten Syntaxbaum, ab. Komponenten dieser Phase nennt man auch Scanner und Parser.

Die **Synthese-Phase** erzeugt, auf Basis des Syntaxbaumes, einen Zwischencode. Mit diesem wird optional die Code-Optimierung durchgeführt und anschließend der Maschinencode erzeugt. (vgl. [CI02], [WC14], [KP13]).

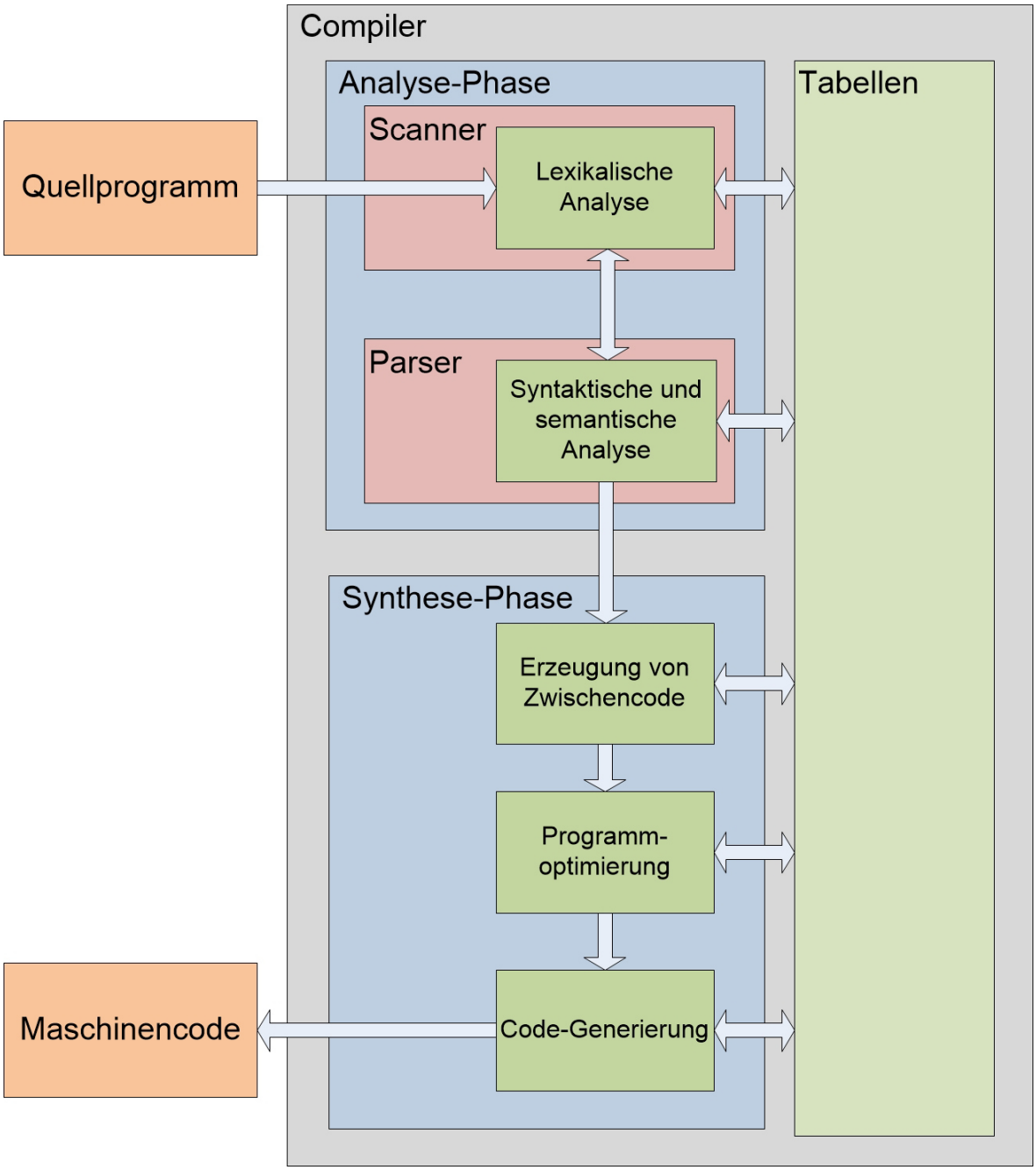

Abbildung 2: Schematischer Aufbau eines Compilers

2.3 Analyse-Phase

2.3.1 Lexikalische Analyse (Scanning)

Die lexikalische Analyse hat zum Ziel, ein Quellprogramm, welches aus einer endlichen Menge von Zeichen eines Zeichenvorrates (z.B. ASCII) besteht, sequentiell zu lesen und in lexikalische Einheiten (Token) aufzuteilen.

Ein Token ist eine Zeichenfolge (Lexem). Dem Token wird, von der Grammatik einer Sprache, ein bestimmter Typ zugewiesen. Der Typ ist die Bezeichnung für einen Token. Die Bezeichnung für einen Token und seine zulässigen Zeichen, werden durch den Entwickler eines Scanners definiert. Grundlage dafür sind die regulären Ausdrücke, die in Kapitel 2.6 näher beschrieben werden.

Das folgende Beispiel, einer Methodendeklaration in Java, zeigt, wie die lexikalische Analyse, eine Zeichenkette in Token zerlegen könnte.

```
public static void main(){}
```

	Token	Type
1	public	Zugriffsmodifizierer
2	static	Zugriffsmodifizierer
3	void	Rückgabetyp
4	main	Methodenname
5	(	öffnende Klammer
6	)	schließende Klammer
7	{	öffnende geschweifte Klammer
8	}	schließende geschweifte Klammer

Abbildung 3: Aufteilung der Token

Die Reihenfolge der erkannten Token wird auch Token-Strom genannt und ist für die syntaktische Analyse von besonderer Bedeutung.

Die Compiler-Komponente, die die lexikalische Analyse durchführt, wird auch Scanner genannt. (vgl [AR], [WTOK14], [WPG14], [KP13])

2.3.2 Syntaktische Analyse (Parsing)

Die Syntax einer Programmiersprache wird durch sogenannte Syntaxregeln beschrieben. Sie werden, auf der Grundlage der kontextfreien Grammatik, festgelegt und mittels der Erweiterten Backus-Naur-Form dargestellt (Kap. 2.5).

Die syntaktische Analyse hat die Aufgabe den erstellten Token-Strom aus der lexikalischen Analyse auf syntaktische Korrektheit zu prüfen. Der Parser, so heißt die Compiler-Komponente, die die syntaktische Analyse durchführt, liest den Token-Strom und zerteilt ihn, unter Beachtung der Syntaxregeln, in syntaktische Einheiten. Ein einzelner Token ist die kleinste syntaktische Einheit. Kombinationen von Token werden zu größeren syntaktischen Einheiten zusammengefasst. Ein Parser sucht, für eine Folge von Token, die größte syntaktische Einheit, welche durch eine Syntaxregel definiert wurde. Die syntaktische Analyse baut durch das Erkennen von syntaktischen Zusammenhängen einen sogenannten Syntaxbaum auf, der durch weitere Komponenten oder Analysen verwendet werden kann. (vgl. [KP13], [AR], [WC14], [WT14])

2.3.3 Semantische Analyse

Die semantische Analyse kann keiner eigenen Compiler-Komponente zugeordnet werden. Sie ist vielmehr ein Bestandteil des Parsers. Die Aufgabe besteht darin, den syntaktisch ausgewerteten Token-Strom auf semantische Korrektheit zu prüfen. In diesem Zusammenhang meint semantisch Korrekt, dass geprüft wird, ob beispielsweise Variablen erst deklariert wurden bevor sie benutzt werden. Dies geschieht, indem den Knoten des erstellten Syntaxbaums Attribute angetragen werden. Diese werden benutzt um Symboltabellen zu erstellen, die anschließend semantisch ausgewertet werden. Dadurch wird erkannt, ob Variablen doppelt deklariert wurden oder einer Variablen ein Wert zugewiesen wurde bevor sie deklariert wurde. Bei objektorientierten Sprachen müssen beispielsweise mehrere solcher Tabellen angelegt werden. Jede dieser repräsentiert dabei einen bestimmten Gültigkeitsbereich. Dieser kann sich beispielsweise nur auf eine einzelne Klasse beziehen. Das bedeutet, dass eine Variable mit dem gleichen Namen in verschiedenen Klassen deklariert sein kann, aber nicht in derselben. Das Ergebnis einer semantischen Analyse nennt man auch attributierter Syntaxbaum. Dieser kann anschließend von weiteren Phasen oder Komponenten zur Weiterverarbeitung genutzt werden. (vgl. [KP13], [AR], [WC14], [WT14], [WCB14])

2.4 Synthese-Phase

Diese Phase soll nur kurz erklärt werden, da sie für die weitere Arbeit keine bedeutende Funktion erfüllt.

Während der Synthese-Phase, wird aus dem attributierten Syntaxbaum ein Zwischencode erzeugt. Dieser ist aber noch nicht auf einer Maschine ausführbar. Er dient lediglich der Code-Optimierung. Optimierungsmöglichkeiten sind beispielsweise die Verbesserung der Laufzeit des Zielprogrammes oder die Minimierung des Speicherplatzbedarfes. Ist die Optimierung abgeschlossen, wird aus diesem Code der Maschinencode erzeugt. Dies geschieht mittels eines Code-Generators. Dieser könnte aber auch direkt aus dem attributierten Syntaxbaum Maschinencode generieren. Die Optimierung ist also optional und muss nicht zwingend durchgeführt werden. (vgl. [KP13], [AR], [WC14], [WT14], [WCB14])

2.5 Erweiterte Backus-Naur-Form

Die Erweiterte Backus-Naur-Form (kurz EBNF) ist eine Metasprache, um kontextfreie Grammatiken darzustellen. Wie diese erstellt werden, beschreibt die Theorie der kontextfreien Grammatik. Mittels der EBNF kann die gesamte Syntax einer Sprache beschrieben werden.

Das folgende Beispiel zeigt die Beschreibung einer Syntaxregel für eine Methodendeklaration (Java) in der EBNF.

```
<method_declaration> ::= {<modifier>}* <type> <identifier> "("  [<parameter_list>] ")"
                         {"["  "]"}* (<statement_block> | ";").


<modifier> ::= "public" | "private" | "protected" | "static" | "final" | "native"
             | "synchronized" | "abstract" | "threadsafe" | "transient".
```

Abbildung 4: Erweiterte Backus-Naur-Form

Eine Syntaxregel (Ersetzungsregel) besteht immer aus einer linken und einer rechten Seite. Das **Nichtterminalsymbol** (Typ eines Token), welches immer einzeln auf der linken Seite stehen muss, wird durch eine Folge von Nichtterminal- und Terminalsymbolen der rechten Seite definiert. **Terminalsymbole** sind Zeichenketten (Token), welche nur auf der rechten Seite einer Syntaxregel benutzt werden dürfen. Sie präsentieren sich selbst als kleinste lexikalische Einheit (vgl. [WKG14], [WEBNF14]).

In Abbildung 4 ist zu erkennen, dass das Nichtterminalsymbol *modifier* durch die Terminalsymbole der rechten Seite definiert wird. Terminalsymbole werden zwischen

Anführungszeichen eingeschlossen. Der vertikale Strich stellt eine Alternative dar. Das heißt, dass ein *modifier* beispielsweise *public* oder auch *private* sein kann.

Wie in der Abbildung 4 zu erkennen, beginnt eine Methodendeklaration mit einem optionalen *modifier*. Optional bedeutet, dass der *modifier* nicht vorkommen muss. Das Symbol „*“ hat die Funktion, das ein Nichtterminalsymbol, welches ihm vorangestellt ist, keinmal oder beliebig oft vorkommen darf. In der Abbildung 3 aus Kapitel 2.3.1 wären *public* und *static* die optionalen *modifier*, die die syntaktische Analyse erkennt und einer Methodendeklaration (syntaktische Einheit) zuordnen kann.

Eine EBNF definiert weitere Sonderzeichen und ihre Bedeutungen. Bezüglich der weiteren Ausführungen in dieser Arbeit ist es aber nicht notwendig, speziell darauf einzugehen. (vgl. [WKG14], [WEBNF14], [CB14], [WT14])

2.6 Reguläre Ausdrücke

Regulärer Ausdrücke werden durch reguläre Grammatiken, welche einer syntaktischen Regel folgen, erzeugt.

Durch die Verwendung der regulären Ausdrücke ist es möglich, ein Nichtterminalsymbol (Type eines Token) durch ein festgelegtes Muster (vorkommen von Zeichen in einer bestimmten Folge) zu beschreiben. Aufgrund dieser Definition, ist ein Scanner in der Lage, eine gelesene Folge von Zeichen mit den regulären Ausdrücken zu vergleichen und bei einer Übereinstimmung dem dazugehörigen Nichtterminal zuzuordnen.

Die folgenden Beispiele zeigen, wie Nichtterminale durch Muster beschrieben werden können.

```
3  number      [0-9]+                % Bsp.   % 365
4  hexnumber   0[xX][0-9a-fA-F]+             % 0xABAD1DEA
5  character   [a-zA-Z]                      % H
6  word        [a-zA-Z]+                     % Eins
```

Abbildung 5: Reguläre Ausdrücke

Das Nichtterminal *hexnumber* muss demzufolge mit einer „0“ beginnen. Darauffolgend ein „x“, welches als Groß- oder Kleinbuchstabe vorkommen kann, und anschließend eine unendliche Menge an Zahlen oder Buchstaben zwischen „a“ und „f“ beziehungsweise „A“ und „F“. Das Symbol „+“ hat die Funktion, dass der Ausdruck, der ihm vorangeht, mindestens einmal vorkommen muss, aber auch beliebig oft vorkommen kann.

(vgl. [WKG14], [WRA14])

2.7 Syntaxbaum

Ein Syntaxbaum wird auch als Ableitungsbaum oder Parse-Baum bezeichnet. Er ist eine baumartige Darstellung der kontextfreien Grammatik einer Sprache. Der Parser baut aus den erkannten syntaktischen Einheiten eine programminterne Struktur auf, welche sich als Syntaxbaum darstellen lässt. Die Wurzel besteht aus dem Startsymbol der Grammatik. In Abbildung 6 ist diese vom Typ *program*. Knoten repräsentieren Nichtterminalsymbole und Blätter Terminalsymbole. Die Nachfolger eines Knoten werden als Kinder bezeichnet. Sie folgen einer festgelegten Reihenfolge. Wie in der Abbildung 6 zu sehen, hat der Knoten *package_header* genau drei Kinder, welche in einer bestimmten Reihenfolge angeordnet sind. Diese Kinder können wieder Knoten anderer Kinder sein. Solang bis das letzte Kind ein Blatt ist.

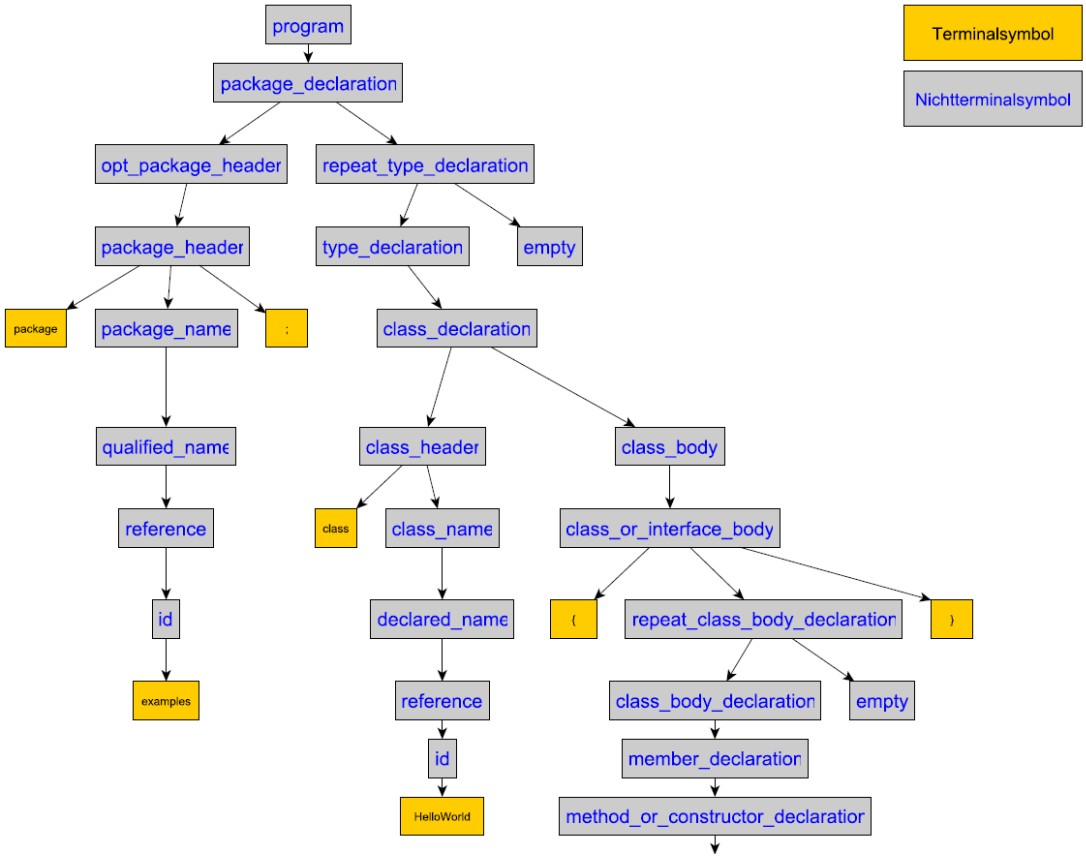

Abbildung 6: Syntaxbaum

Liest man die Terminalsymbole von links nach rechts, so erhält man den Token-Strom der von einem Scanner gelesen und von einem Parser syntaktisch erkannt wurde. Leerzeichen, Zeilenumbrüche und Tabulatoren sind Zeichen, die die lexikalische Analyse als Trennzeichen erkennen kann. Das bedeutet, dass sie nicht als gültige Token der Sprache akzeptiert werden.

package examples ; class HelloWorld { }

(vgl. [WS14])

3 Generatoren

Wie im vorangegangen Kapitel erklärt, besteht ein Compiler aus verschiedenen Komponenten die sich in Phasen unterteilen lassen. Jede dieser hat eine spezielle Aufgabe, wie beispielsweise die lexikalische oder syntaktische Analyse. In diesem Kapitel soll aufgezeigt werden, welche Möglichkeiten es gibt, einzelne Bestandteile eines Compilers oder einen vollständigen Transcompiler automatisiert zu erstellen und welche Werkzeuge dafür genutzt werden können.

3.1 Scanner-Generator

Wie bereits bekannt, ist ein Scanner für die lexikalische Analyse in einem Compiler zuständig. Damit man diese Komponente nicht selbst programmieren muss, gibt es sogenannte Scanner-Generatoren. Diese Programme erzeugen unter Verwendung einer lexikalischen Beschreibung der Quellsprache einen Scanner, der in der Lage ist Quellcode zu lesen und in zulässige Token aufzuteilen. Bekannte Scanner-Generatoren sind unter anderem Lex und Flex. (vgl. [WLEX14], [WC14], [WCB14], [WTOK14])

3.2 Parser-Generator

Für die syntaktische Analyse gibt es sogenannte Parser-Generatoren. Dies sind Programme die unter Verwendung einer syntaktischen Beschreibung der Quellsprache und einem Scanner, Programme erzeugen, welche als Grundlage im Compilerbau genutzt werden können. Typische Vertreter für Parser-Generatoren sind Yacc und Bison. (vgl. [WYACC14], [WC14], [WCB14], [WP14], [WPG14], [WGB14]])

3.3 Transcompiler-Generator

Ein Transcompiler ist ein Compiler, welcher Quellcode einer Sprache in Quellcode einer anderen Sprache transformiert. Er erzeugt keinen Maschinencode und keine ausführbaren Dateien die von einer Maschine gelesen werden können. Der Begriff Transcompiler-Generator existiert nicht. Er soll aber in diesem Zusammenhang benutzt werden, um deutlich zu machen, dass es Programme gibt, welche in der Lage sind, unter Verwendung einer lexikalischen und syntaktischen Beschreibung einer Quellsprache und

einer Beschreibung von Transformationsregeln, ein Programm zu erzeugen, welches den Quellcode einer Sprache in den einer anderen transformiert. Ein Beispiel für einen Transcompiler-Generator ist die *Turing eXtender Language* (kurz TXL). (vgl. [WC14], [WCB14], [TXL13], [CDMS02])

3.4 Zusammenfassung des Kapitels

In diesem Kapitel wurde erklärt, welche Möglichkeiten es gibt, einen vollständigen Transcompiler oder nur Teile eines Compilers automatisiert zu erzeugen.

Flex und Bison sind Generatoren, welche bei einer Transformation von Quellcode nur den Teil der Analyse-Phase übernehmen können. Eine Transformation des generierten Syntaxbaumes ist, mit Flex und Bison, nicht möglich. Diesbezüglich werden die beiden Generatoren, in dieser Arbeit nicht weiter untersucht.

Im Folgenden soll untersucht werden, ob mit der *Turing eXtender Language* die Erstellung eines Compilers zur Transformation von Quellcode möglich ist.

4 Turing eXtender Language

Im vorangegangenen Kapitel wurde erklärt, dass es Generatoren gibt mit denen einzelne Komponenten eines Compilers oder vollständige Transcompiler erzeugt werden können.

In diesem Kapitel wird das ausgewählte syntaxbasierte Werkzeug näher beschrieben. Es wird die prinzipielle Arbeitsweise erklärt und es wird untersucht ob eine Transformation von Quellcode mit diesem möglich ist.

4.1 TXL

Die Turing eXtender Language (kurz TXL) ist ein Compiler / Interpreter der im Jahr 1985 von Charles Halpern-Hamu und James Reginald Cordy an der Universität von Toronto erforscht und entwickelt wurde. Die eigentliche Idee, die mit TXL verfolgt wurde, war, wie es der Name schon sagt, die Programmiersprache „Turing“ weiter zu entwickeln. TXL sollte zum Experimentieren mit Programmiersprachen-Notationen und zur Erweiterung von Programmiersprachen und Dialekten dienen.

TXL wurde in den letzten Jahren stetig weiterentwickelt und wird inzwischen für viele verschieden Zwecke eingesetzt.

- Sourcecode analysis
- System migration
- Code generation from models
- UML model extraction and transformation
- Clone detection
- …

Unter anderem hat sich TXL auch zu einem Transformations-System für Programmtransformationen entwickelt. Diesbezüglich wird TXL heutzutage auch als „Tree-Transformation-Language“ bezeichnet.

FreeTXL ist das offizielle Tool, welches durch das TXL Projekt der Queen´s Universität unterstützt wird. Es steht für verschiede Plattformen zur Verfügung und darf uneingeschränkt genutzt werden. FreeTXL (im Folgenden TXL genannt) ist ein XML-basierender Compiler / Interpreter für die verschiedensten Arten von Transformationsaufgaben.

TXL ist ein in sich geschlossener Compiler / Interpreter. Das bedeutet, dass alle Komponenten die für einen solchen benötigt werden, wie beispielsweise ein Scanner,

Parser oder Code-Generator, bereits implementiert sind. Es werden also keine externen Parser, Frameworks oder Bibliotheken benötigt. (vgl. [TXL13], [CDMS02], [ETC11], [TPL12], [UGTCI12])

4.1.1 Prinzipielle Arbeitsweise der TXL

Die TXL hat eine funktionale Grundstruktur, welche durch Term-Ersetzungsregeln erweitert wird. Transformationen basieren auf Term-Ersetzungen in Bäumen. Diese werden jedoch durch beispielbasierte Regelspezifikationen, ähnlich der EBNF, überwiegend vor dem Programmierer versteckt. Die folgende Abbildung veranschaulicht die einzelnen Phasen, welche während eines Transformationsprozesses mit der TXL durchlaufen werden.

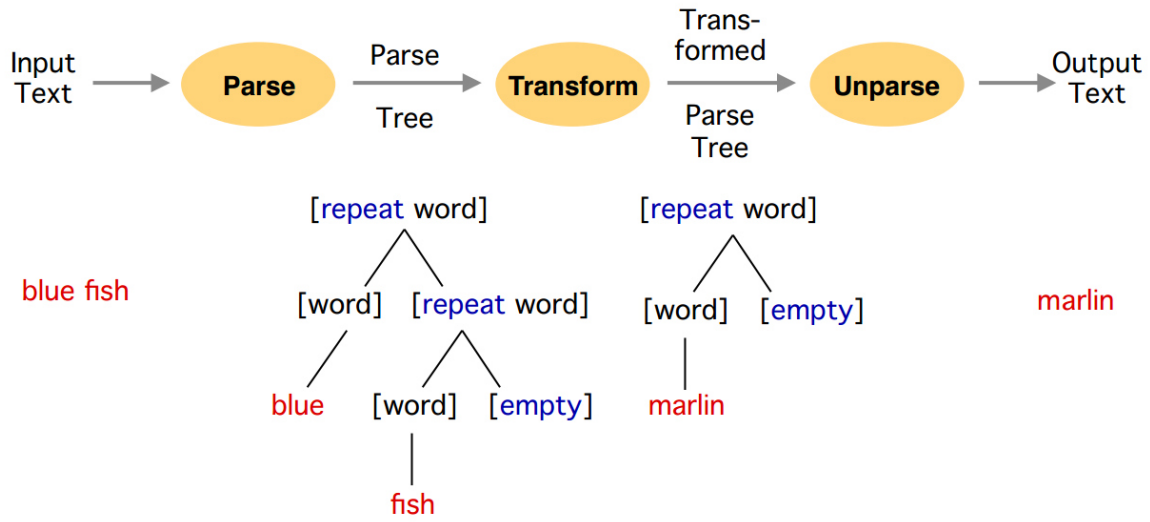

Abbildung 7:Prinzipielle Arbeitsweise der TXL

Der Prozess der Transformation unterteilt sich in drei Phasen.

1. Parse
2. Transform
3. Unparse

In der ersten Phase wird der zu transformierende Quelltext gelesen und mittels einer Grammatikbeschreibung analysiert. Während der Analyse werden die erkannten Token in einen strukturierten Baum (Syntaxbaum) geschrieben.

In der zweiten Phase wird dieser Baum unter Verwendung der Transformationsregeln auf Muster (Quell-Pattern) untersucht. Bei einer Übereinstimmung wird das Quell-Pattern durch das Ziel-Pattern ersetzt. Dabei entsteht ein neuer Baum (transformierter Syntaxbaum) der dann der Zielsprache entspricht.

Die dritte Phase hat die Aufgabe den neu erstellten Baum strukturiert auszuwerten und in den Quelltext der Zielsprache zu schreiben.

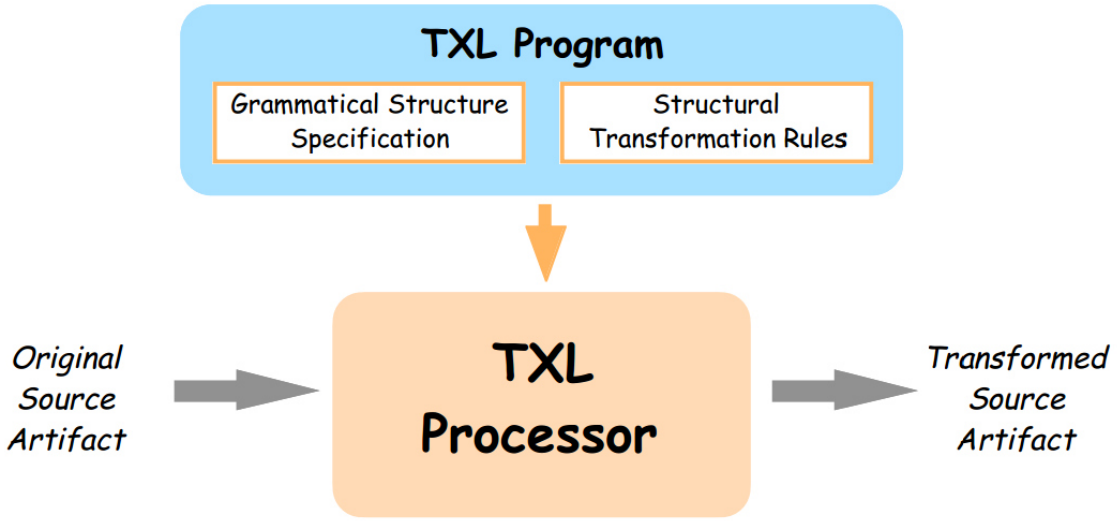

Abbildung 8: Zusammenwirken TXL-Transformation

Damit eine TXL-Transformation durchgeführt werden kann, benötigt der TXL-Interpreter ein TXL-Programm, sowie den Quelltext, welcher übersetzt werden soll. Die Abbildung 8 zeigt eine vereinfachte Darstellung des Zusammenwirkens dieser Bestandteile bei einer Transformation. (vgl. [TXL13], [CDMS02], [ETC11], [TPL12], [UGTCI12])

4.1.2 Struktureller Aufbau eines TXL-Programmes

Ein TXL-Programm setzt sich aus mindestens zwei Bestandteilen zusammen. Einer Grammatikbeschreibung der Quellsprache und einer Beschreibung der Transformationsregeln. TXL-Programme werden durch eine kontextfreie Grammatik, vergleichbar mit der EBNF, beschrieben. Die lexikalische und syntaktische Form einer Quellsprache wird bei TXL in einer einzigen Datei vereint. Der Name dieser Datei ist in der Regel der der Programmiersprache, welche transformiert werden soll und endet mit der Erweiterung *„Grm“* (Bsp.: *„Java.Grm“*). Die Transformationsregeldateien besitzen die Erweiterung *„Rul“* und beschreiben die einzelnen Regeln, welche für eine Transformation benötigt werden (Bsp.: *„TranslateMembers.Rul“*). Ein TXL-Programm, mit der Erweiterung *„Txl“* (Bsp.: *„JavaToC#.Txl“*), implementiert alle Grammatik- und Regeldateien durch

sogenannte *„include“* Anweisungen. Die Folgende Abbildung verdeutlicht den strukturellen Zusammenhang der einzelnen Bestandteile.

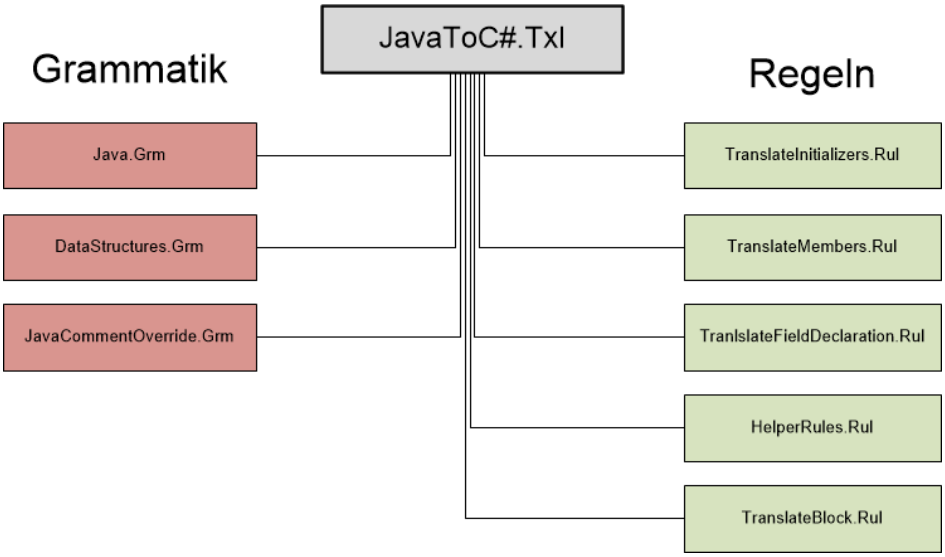

Abbildung 9: Aufbau eines TXL-Programmes

Wie in der Abbildung 9 zu sehen, kann ein TXL-Programm aus mehreren Regel- und Grammatikdateien bestehen. Es ist also möglich, Regeln für bestimmte Transformationsaufgaben zu gruppieren und in separate Regeldateien aufzuteilen. Dies erhöht die Übersicht des TXL-Programmes und macht es dadurch, für einen Programmierer, besser lesbar. (vgl. [TXL13], [CDMS02], [ETC11], [TPL12], [UGTCI12], [EEA05])

4.2 Zusammenfassung des Kapitels

Die Geschichte der Turing eXtender Language und ihr Funktionsumfang haben gezeigt, dass sie für die Transformation von Quellcode bestens geeignet ist.

Diesbezüglich wird sie in den folgenden Kapitelen näher untersucht und beschrieben.

5 Grundlagen eines TXL-Programmes

In diesem Kapitel werden die Grundlagen für die Programmierung eines TXL-Programmes näher erklärt. (vgl. [TXL13], [CDMS02], [ETC11], [TPL12], [UGTCI12])

5.1 Grundlagen für die Beschreibung der Grammatikdatei

Dieses Unterkapitel erklärt die Grundlagen einer Grammatikbeschreibung mit der TXL. Anhand von Beispielen wird gezeigt, welche Elemente es gibt und wie sie genutzt werden.

5.1.1 Terminal- und Nichtterminalsymbole

Die Basis für eine Grammatikbeschreibung mit der TXL, bildet das *define* Statement. Es ist ähnlich aufgebaut, wie eine Syntaxregel in der EBNF (Kap. 2.5). Das folgende Beispiel zeigt die Definition eines Nichtterminalsymbols.

```
6  define expression                    % Definition Nichtterminalsymbol
7          [number]                     % Verwendung eines Nichtterminalsymbol
8      |   [expression] + [number]      % Kombination von Nichtterminalsymbolen ...
9      |   [expression] - [number]      % ... als Alternative
10 end define
```

Abbildung 10: TXL define

In Zeile 6 wird der Name eines Nichtterminalsymboles festgelegt. Der Name repräsentiert den Typ eines solchen. In diesem Beispiel ist das Nichtterminal vom Typ *expression* (Ausdruck). Zeile 7 beschreibt, durch welches Nichtterminalsymbol, eingeschlossen in eckigen Klammern, *expression* ersetzt werden kann. In diesem Fall durch *number*. Jedes Nichtterminal muss mindestens einmal durch eine *define* Anweisung definiert werden, damit es in seinem eigenen oder in einem anderen Definitionsblock benutzt werden kann. Der vertikale Strich, in Zeile 8 und 9, symbolisiert, dass es für *expression* auch noch Alternativen zu *number* gibt. Eine *expression* kann demzufolge auch eine *expression* selbst, gefolgt von einem Additionsoperator und einer *number* sein. Zeile 10 schließt den Definitionsblock, mittels *end define*, wieder ab. Ausdrücke wie 5, 17+8 oder 4-6+9 sind demnach zulässige Zeichenketten vom Typ *expression*.

Das *redefine* Statement bietet die Möglichkeit, eine schon existierende Definition zu erweitern. Sollte beispielweise eine neue Version von Java erscheinen, so ist man in der

Lage, die Grammatikbeschreibung experimentell mit den neuen Inhalten zu erweitern, ohne die existierenden Definitionen ändern zu müssen.

Im folgenden Beispiel wird die Definition von *expression*, durch zusätzliche Kombinationen von Nichtterminalen, ergänzt.

```
14  redefine expression                  % Erweiterung der Definition
15          ...                          % Benutzung des alten Definitionsblockes
16      |   [expression] * [expression]  % zusätliche Kombination ...
17      |   [expression] / [expression]  % ... als Alternative
18  end redefine
```

Abbildung 11: TXL redefine

In Zeile 15 wird durch drei Punkte angegeben, dass der Definitionsblock aus dem define Statement von *expression* erweitert werden soll. Ohne die Punkte würde die Definition von *expression* überschrieben werden.

```
64  define program                       %Definition Nichtterminalsymbol program
65      [package_declaration]            %Verweis auf package_declaration
66  end define                           %Definitionsende
```

Abbildung 12: TXL define

Die Definition des Nichtterminalsymboles *program* bildet den Startpunkt einer Grammatikbeschreibung. Dieses muss definiert werden, da es das erste Nichtterminal ist auf das ein TXL-Programm zugreift.

Das *keys* Statement in einer Grammatikbeschreibung wird genutzt, um Schlüsselwörter einer Quellsprache festzulegen. Diese werden durch ein Leerzeichen voneinander getrennt, aufgezählt. Sollten Schlüsselwörter der Quellsprache mit denen der TXL übereinstimmen, müssen diese mit einem vorangestellten Apostroph versehen werden. Die Abbildung 13 zeigt, das *function*, *repeat* und *end* in beiden Sprachen Schlüsselwörter sind und deshalb erweitert werden müssen

```
10  keys
11      program procedure
12      function repeat until
13      for while do begin end
14  end keys
```

```
16  keys
17      program procedure
18      'function 'repeat until
19      for while do begin 'end
20  end keys
```

Abbildung 13: TXL keys

Das Nichtterminalsymbol *[key]* ermöglicht anschließend den Zugriff auf die Schlüsselwörter, welche im *keys* Statement definiert wurden.

Das *tokens* Statement bietet die Möglichkeit, eigene Nichtterminalsymbole, auf Grundlage der regulären Ausdrücke (Kap.: 2.6) zu definieren.

```
60  tokens
61      number   "\d+(.\d+)?([eE][+-]?\d+)?"
62  end tokens
```

Abbildung 14: TXL tokens

Abbildung 14 zeigt die Definition des Nichtterminals *number*. Die regulären Ausdrücke beschreiben die zulässigen Zeichen und deren Reihenfolge für das Nichtterminal. Dieser Ausdruck muss in Anführungszeichen eingeschlossen sein. Damit eine Zeichenkette dem Nichtterminal vom Typ *number* zugewiesen werden kann, muss es dem Muster entsprechen. Eine zulässige Zeichenkette würde mit einer Zahle zwischen „0“ und „9“ beginnen. Diese können auch mehrfach vorkommen. Darauf folgend ein optionaler Punkt mit weiteren Zahlen. Anschließend wieder ein optionaler Block von Zeichen. Dieser Block muss mit einem Buchstaben „e“ oder „E“ beginnen, gefolgt von einem optionalen „+“ oder „-“ Zeichen und am Ende mindestens mit einer Zahl enden.

5.1.2 Vordefinierte Nichtterminalsymbole

Die TXL bietet vordefinierte Nichtterminalsymbole, welche für die Ersterstellung einer Grammatik genutzt werden können. Darüber hinaus können, wenn dies erforderlich ist, auch eigene Nichtterminale, durch die Beschreibung mit regulären Ausdrücken (Kap. 2.6), im *tokens* Statement, definiert werden.

Nichtterminalsymbol	Erklärung
[id]	Jede Zeichenfolge beginnend mit einem Buchstaben oder Unterstrich. Bsp.: ABC, abc, a9b56, _ab5, Ab_C_
[upperlowerid]	Einschränkung von [id]. Muss mit einem Großbuchstaben beginnen. Bsp.: AbcD, A_bcd
[upperid]	Einschränkung von [id].Darf nur Großbuchstaben beinhalten. Bsp.: ABCD
[lowerupperid]	Einschränkung von [id]. Muss mit einem Kleinbuchstaben beginnen. Bsp.: abcD, v_bcd
[number]	Jede Zahl ohne Vorzeichen, gefolgt von weiteren optionalen Zahlen oder einem Dezimalpunkt, gefolgt von weiteren Zahlen und einem optionalen Exponent (e, E). Bsp.: 123, 12.34, 123.45e22
[floatnumber]	Einschränkung von [number]. Muss einen Exponenten enthalten. Bsp.: 12.3e22, 1E2
[decimalnumer]	Einschränkung von [number]. Muss einen Dezimalpunkt enthalten. Bsp.: 12.3, 1.0
[integernumber]	Einschränkung von [number]. Darf weder Exponent noch Dezimalpunkt beinhalten. Bsp.: 12, 2, 9523
[stringlit]	Jede Zeichenfolge, die mit Anführungszeichen beginnt und endet. Entwertete Anführungszeichen werden nicht als Endzeichen erkannt. Bsp.: "Hello World!", "Hello \"World! \" "
[charlit]	Wie [stringlit]. Nur mit einfachen Anführungszeichen. Bsp.: 'Hello World!', 'Hello \'World! \' '
[comment]	Kommentare die im *comments* statement definiert wurden. Bsp.: //, /* */
[key]	Alle Schlüsselwörter einer Sprache, welche in dem *keys* statement definiert wurden.
[token]	Alle, durch reguläre Ausdrücke, selbstdefinierten Nichtterminale im *tokens* statement.
[srclinenumber]	Schreibt die Zeilennummer des Quellcodes, wo ein Muster erkannt wurde, in das Zielprogramm.
[srcfilename]	Schreibt den Namen der Quelldatei, welche benutzt wurde, an die entsprechende Stelle im Zielprogramm.

Abbildung 15: TXL Vordefinierte Nichtterminalsymbole

5.1.3 Modifikatoren für Nichtterminalsymbole

Alle Nichtterminalsymbole, eingeschlossen in eckigen Klammern, können, durch bestimmte Schlüsselwörter der TXL, modifiziert werden. Wie in Abbildung 16 zu sehen, wird das Nichtterminal *package_declaration* durch eine festgelegte Reihenfolge von Nichtterminalsymbolen beschrieben.

```
39  define package_declaration          %Definition Nichtterminalsymbol
40      [opt package_header]            %optional package_header
41      [repeat import_declaration]     %mehrfach import_declaration
42      [repeat type_declaration+]      %mehrfach type_declaration
43  end define                          %Definitionsende
```

Abbildung 16: TXL Modifikatoren

Der Modifikator *opt* hat dabei die Funktion das *package_header* nicht zwingend Bestandteil des Musters sein muss. Er ist also optional.

Mit *repeat* wird erreicht, das beispielsweise das Nichtterminal *import_declaration* mehrfach, aber auch gar nicht vorkommen kann. Ist es aber erforderlich, das ein mit *repeat* erweitertes Nichtterminal mindestens einmal vorkommen muss, so ergänzt man es mit dem „+“ Symbol.

Bei der Definition eines Nichtterminalsymboles, sollte daher immer darauf geachtet werden, dass dieses niemals auf einen Operator endet.

```
46  define constructor_declaration      %Definition constructor_declaration
47      [repeat modifier]               %mögliche modifier
48      [constructor_declarator]        %Zugriff auf die Def. constructor_declarator
49      [opt throws]                    %optional throws
50      [constructor_body]              %Zugriff auf die Def. constructor_body
51  end define                          %Definitionsende
52
53  define constructor_declarator       %Definition constructor_declarator
54      [class_name]                    %Zugriff auf die Def. class_name
55      '(                              %Terminalsymbol öffnende Klammer
56          [list formal_parameter]     %kommaseparierte Liste aller formal_parameter
57      ')                              %Terminalsymbol schließende Klammer
58  end define                          %Definitionsende
```

Abbildung 17: TXL Modifikatoren

Der *list* Modifikator hat die Funktion, dass das folgende Nichtterminal mehrfach oder gar nicht vorkommen darf. Sollte es mehrfach vorkommen, so werden die Elemente in einer kommaseparierten Liste gespeichert. Wie in Abbildung 17 zu sehen, können bei einem Konstruktor mehrfach formale Parameter vorkommen, welche mittels *list* in eine durch Komma getrennte Liste aufgenommen werden.

5.2 Grundlagen für die Beschreibung der Regeldatei

Dieses Unterkapitel erklärt die Grundlagen, wie eine Regeldatei mit der TXL beschrieben wird. Anhand von Beispielen wird gezeigt, welche Elemente es gibt und wie sie benutzt werden.

5.2.1 Funktionen und Regeln

Mittels Funktionen und Regeln besteht die Möglichkeit, Teile eines Parse-Baumes auf vorgegebene Muster zu untersuchen und durch neue Muster zu ersetzen. Der wichtigste Unterschied zwischen diesen beiden ist, das eine Funktion nur das erste Vorkommen eines Suchmusters ersetzt. Wohingegen Regeln jedes Vorkommen ersetzen.

Die Abbildung 18 zeigt den Aufbau einer Funktion und ihre Bestandteile.

```
68  function name                %Definition der Funktion
69      replace [type]           %Typ des Teilbaumes
70          pattern              %Suchmuster
71      by                       %Ersetzungsanweisung
72          replacement          %Ersetzungsmuster
73  end function                 %Definitionsende
```

Abbildung 18: TXL Funktion

In Zeile 68 wird die Funktion definiert und mit einem Funktionsnamen beschrieben. Über diesen Namen wird die Funktion eindeutig erkannt und kann in einem TXL-Programm angesprochen werden. In Zeile 69 wird, durch die Angabe eines Nichtterminals, festgelegt, von welchem Typ der Knoten sein muss, den die Funktion im Parse-Baum sucht. Dieses Nichtterminal bildet, bei einer erfolgreichen Suche, das Wurzelelement für das Suchmuster, welches in Zeile 70 angegeben wird. Kommt es dabei zu einer Übereinstimmung, wird dieses durch das Ersetzungsmuster aus Zeile 72 ersetzt.

Die Benutzung einer Funktion kann wie folgt beschrieben werden. In der Mathematik schreibt man für eine Funktion *f(x)*. Wobei *x* eine unabhängige Variable ist. In der TXL wird eine Funktion nach dem Muster *x[f]* benutzt. Die Variable *x* repräsentiert dabei eine Referenz auf den Knoten des Teilbaumes, welche ihr vorher zugewiesen wurde. Die Funktion *[f]* bezieht sich demzufolge nur auf diesen Teilbaum. Bei einem Aufruf der Funktion aus Abbildung 19 (Funktionsaufruf: *x[changePackageToNamespace]*) würde dies bedeuten, dass die Funktion prüft, ob in dem Teilbaum auf den die Variable x verweist, ein Knoten vom Typ opt package_header existiert. Eine Funktion prüft nur solang die Existenz eines Knotens, bis dieser gefunden wurde. Sollte der Knoten mehrfach vorhanden sein, wird dies von einer Funktion nicht berücksichtigt.

Abbildung 19 zeigt eine Funktion, welche die Aufgabe hat, das Muster einer package-Deklaration in Java durch das Muster einer namespace-Deklaration in C# zu ersetzen.

```
75  function changePackageToNamespace
76      replace [opt package_header]
77          'package Name[package_name] ';
78      by
79          'namespace Name
80  end function
```

Abbildung 19: TXL Funktion

Wenn der Knoten in dem Teilbaum gefunden wurde, wird geprüft, ob das Suchmuster, bestehend aus einem Terminalsymbol *package*, gefolgt von einem Nichtterminal *package_name* und einem Terminalsymbol ***;*** , unmittelbar unterhalb des Knotens vorhanden ist. Eine Funktion, so wie sie in Abbildung 19 gezeigt wird, sucht das Muster als seine direkten Kindelemente. Sie durchsucht nicht den kompletten Teilbaum. Kommt es dabei zu einer Übereinstimmung, wird dieses Muster durch das Ersetzungsmuster in Zeile 79 ersetzt. Die Variable *Name* hat dabei die Funktion, den Teilbaum vom Typ *package_name* in das Ersetzungsmuster zu übernehmen, da dieser unverändert übernommen werden soll.

Abbildung 20 zeigt eine Suchfunktion. Dies wird durch das Symbol * in Zeile 85 erreicht. Mit dieser Funktion ist es möglich, das erste Vorkommen eines Suchmusters im gesamten Teilbaum zu suchen und durch das Ersetzungsmuster zu ersetzen.

```
84  function changePackageToNamespace
85      replace * [opt package_header]
86          'package Name[package_name] ';
87      by
88          'namespace Name
89  end function
```

Abbildung 20: TXL Suchfunktion

Regeln besitzen die gleiche Syntax wie eine Funktion. Dabei wird nur das Schlüsselwort *function* durch das Wort *rule* ersetzt. Im Vergleich zu einer Funktion, sucht eine Regel im gesamten Teilbaum nach dem vorgegebenen Nichtterminal. Kommt es zu einer Übereinstimmung, wird dieser Teilbaum separat auf das Suchmuster untersucht und gegebenenfalls durch das Ersetzungsmuster ersetzt. Anschließend wird dieser neu konstruierte Teilbaum in den alten Teilbaum, auf den die Variable referenziert, eingehangen. Die Regel beginnt von vorn und sucht nach weiteren Knoten mit der Bezeichnung des Nichtterminals. Dies wird solang fortgesetzt bis kein solcher Knoten mehr gefunden wurde.

5.2.2 Variablen

Variablen sind Referenzen auf einen ihnen zugewiesenen Teilbaum. Sie haben einen, vom Programmierer, festgelegten Name. Darauf folgt, in eckigen Klammern eingeschlossen, der Name des Nichtterminalsymboles auf den die Variable referenzieren soll.

```
18  function main
19      replace [program]
20          PHeader[opt package_header]
21          ImportDeclaration[repeat import_declaration]
22          TypeDeclr[repeat type_declaration]
```

Abbildung 21: TXL Variablen

Die Variable *PHeader*, aus Abbildung 21, verweist demzufolge auf den Teilbaum mit dem Nichtterminalsymbol *package_header*.

Mit globalen Variablen können Tabellen angelegt werden. Wie in den Abbildungen zu sehen, wird ein Nichtterminal mit dem Namen *StmtMapper* definiert, welches aus einer Folge von *table_entry* besteht. In der Anweisung *export*, vom Typ StmtMapper, wird diese Tabelle gefüllt. Durch die Anweisung *import*, wird diese Tabelle zur weiteren Verarbeitung breitgestellt.

```
109    define StmtMapper
110        [repeat table_entry]
111    end define

103    define table_entry
104        [entries] '-> [entries]
105    end define
```

```
33        export StatementMapping [StmtMapper]
34        'System.out.println -> 'Console.WriteLine
35        'System.out.print -> 'Console.Write
```

```
406     import StatementMapping [StmtMapper]
```

Abbildung 22: TXL Globale Variablen

5.2.3 Konstruktoren und Dekonstruktoren

Dekonstruktoren werden genutzt um die Referenz einer Variablen in seine Bestandteile zu zerlegen.

```
136    function changeImportToUsing importDec[import_declaration]
137        deconstruct importDec
138            'import Name[package_or_type_name] DotStar[opt dot_star] ';
```

Abbildung 23: TXL Dekonstruktor

Die Abbildung 23 zeigt eine Funktion, welcher eine Variable übergeben wurde. Diese Variable referenziert den Teilbaum mit dem Nichtterminal *import_declaration*. Mittels des Dekonstruktors ist es möglich den Teilbaum in seine Bestandteile zu zerlegen und neuen Variablen zuzuweisen. Mit diesen neuen Variablen kann, innerhalb der Funktion, gearbeitet werden.

Konstruktoren bilden, aus einer Folge von Terminal- und Nichtterminalsymbolen oder Variablen, einen neuen Teilbaum. In der vorliegenden Abbildung 24 besteht der neue Baum aus dem Wurzelelement *program*. Der Konstruktor hat den Namen *NewProgram*. Über diesen Namen kann der Baum, zur weiteren Verarbeitung, angesprochen werden.

```
66        construct NewProgram[program]
67            NewImportDeclaration
68            PHeader[changePackageToNamespace]
69            TypeDeclr[changeClassHeader][changeInterfaceHeader]
```

Abbildung 24: TXL Konstruktor

5.3 Grundlagen der Beschreibung der Programmdatei

Dieses Unterkapitel erklärt die Grundlagen, wie eine Programmdatei mit der Erweiterung *.Txl* beschrieben wird. Anhand von Beispielen wird gezeigt, welche Elemente es gibt und wie sie benutzt werden.

Eine Programmdatei importiert alle erstellten und für die Transformation benötigen Dateien. Dies geschieht mittels einer Anweisung mit dem Namen *include*. Abbildung 25 zeigt ein Beispiel. Die Reihenfolge, wie die Dateien importiert werden, hat dabei keine Bedeutung.

```
6   % The base Java grammar and grammar overrides
7   include "Java.Grm"
8   include "JavaCommentOverrides.Grm"
9   % Translation rule sets
10  include "DataStructures.Grm"
11  include "HelperRules.Rul"
12  include "TranslateMembers.Rul"
13  include "TranslateInitializers.Rul"
14  include "TranslateFieldDeclaration.Rul"
15  include "translateBlockStatements.Rul"
```

Abbildung 25: TXL include

Die wichtigste Funktion oder Regel, welche in einer Programmdatei existieren muss, ist die Funktion mit dem Namen *main*. Diese wird, bei einer Transformation, als erstes aufgerufen.

```
17  % Main translation rule
18  function main
19      replace [program]
20          PHeader[opt package_header]
21          ImportDeclaration[repeat import_declaration]
22          TypeDeclr[repeat type_declaration]
23          %include the System namespace
24          construct DefaultImportDeclaration[repeat import_declaration]
25              'using 'System ;
26          %change each import
27          construct NewImportDeclaration[repeat import_declaration]
28             DefaultImportDeclaration[changeImportToUsing each ImportDeclaration]
29          % before adding braces
30          construct NewProgram[program]
31              NewImportDeclaration
32              PHeader[changePackageToNamespace]
33              TypeDeclr[changeClassHeader][changeInterfaceHeader]
34      by
35              NewProgram[addBraces]
```

Abbildung 26: TXL Funktion main

6 Transformationen mit der TXL

Als Basis für die in diesem Kapitel gezeigten Beispiele dient die Masterarbeit [EEA05] von Mohammad El-Ramly, Rihab Eltayeb und Hisham Allan. Die Autoren haben mittels der TXL experimentell versucht, Java-Quellcode in C#-Quellcode zu transformieren. Als Grundlage für ihre Arbeit diente die Beschreibung der Grammatik für die Version Java 1.1. Sie kann auf der Webseite von TXL [TXL13] heruntergeladen werden. In diesem Kapitel wird erklärt, wie ein TXL-Programm bei der Transformation eines Quelltextes vorgeht, welche Funktionen und Regel benutzt werden und wie der Syntaxbaum transformiert wird.

6.1 Struktur des TXL-Programmes

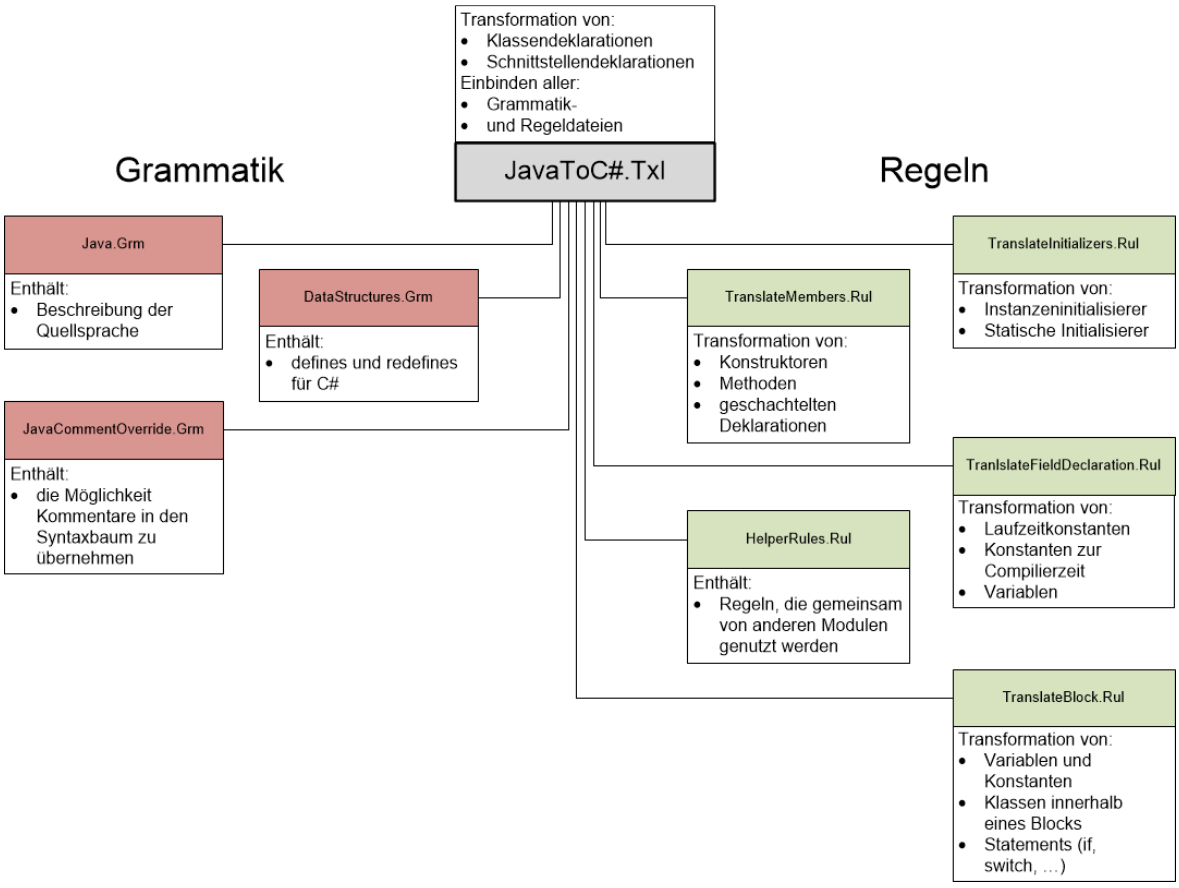

Abbildung 27: TXL Struktur JavaToC#

Die Abbildung 27 zeigt den Aufbau des TXL-Programmes, welches Java-Quellcode nach C#-Quellcode transformiert. Dieses besteht dabei aus mehreren Regel- und Grammatikdateien. Ähnlich wie auch in aktuellen Programmiersprachen Programme auf mehrere Klassen verteilt wird. Jede Regeldatei stellt transformationsspezifische Funktionen und Regel zur Verfügung. Diese Aufteilung macht das Programm, für einen Programmierer, besser les- und erweiterbar. Im Folgend werden die einzelnen Dateien näher erklärt und mit einem kurzen Quellcodeausschnitt visualisiert. Der gesamte Quellcode, zu diesem TXL-Programm, befindet sich im Anhang dieser Arbeit.

6.1.1 Java.Grm

Diese Datei enthält die Beschreibung der Quellsprache Java. Sie wird genutzt um die lexikalische und syntaktische Analyse der Quelldatei durchzuführen.

```
66  % Syntax of Java 1.1
67
68  define program
69      [package_declaration]
70  end define
71
72
73  % Declarations
74
75  define package_declaration
76      [opt package_header]
77      [repeat import_declaration]
78      [repeat type_declaration]
79  end define
80
81  define package_header
82      'package [package_name] '; [NL][NL]
83  end define
84
85  define package_name
86      [qualified_name]
87  end define
88
89  define import_declaration
90      'import [imported_name] '; [NL][NL]
91  end define
```

Abbildung 28: TXL Java.Grm

6.1.2 DataStructures.Grm

DataStructures.Grm beschreibt einige wichtige Erweiterungen der Java.Grm in Bezug auf die Zielsprache C#. Beispielsweise wird die Definition von *modifier*, aus der Java.Grm durch ein *redefine*, um die Modifizierer der Sprache C# erweitert. Darüber hinaus werden weitere C# spezifische Sprachelemente definiert, welche für eine Transformation benötigt werden.

```
12  % for using in class and interface body section
13  % C# seperates the modifiers applied to classes
14  % from those to interfaces and methods
15  redefine modifier
16      ... % java
17      |[class_modifier]      % C#
18      |[interface_modifier] % C#
19      |[method_modifier]    % C#
20      |[constant_modifier]  % C#
21  end redefine
22  % new C# set of modifiers
23  define interface_modifier
24      'new
25      |'public
26      |'protected
27      |'internal
28      |'private
29  end define
30  % new C# set of modifiers
31  define class_modifier
32      'new
33      |'public
34      |'protected
35      |'internal
36      |'private
37      |'abstract
38      |'sealed
39  end define
```

Abbildung 29: TXL DataStructures.Grm

6.1.3 JavaCommentOverrides.Grm

Die JavaCommentOverrides.Grm dient der Erweiterung der Java.Grm. Sie ermöglicht, dass Kommentare des Java-Quellcodes nach einer Transformation erhalten bleiben.

```
12  % Can have comments as declarations
13  redefine class_body_declaration
14      ...
15  |   [comment_NL]
16  end redefine
17
18  redefine package_declaration
19      [repeat comment_NL] ...
20  end redefine
21
22  redefine import_declaration
23      ...
24  |   [comment_NL]
25  end redefine
26
27  redefine type_declaration
28      ...
29  |   [comment_NL]
30  end redefine
31
32  % Can have comments before initializers ...
33  redefine variable_initializer
34      [repeat comment_NL] ...
35  end redefine
36
37  % ... or after initializers ...
38  redefine variable_initializer
39      ... [repeat comment_NL]
40  end redefine
```

Abbildung 30. TXL JavaCommentOverrides.Grm

6.1.4 TranslateInitializers.Rul

Die in dieser Datei zur Verfügung gestellten Funktionen und Regeln haben beispielsweise die Aufgabe, statische Initialisierungen (Java) in statische Konstruktoren (C#) zu transformieren. Wie in Abbildung 31 (Zeile 144) zu sehen, wird bei einer solchen Transformation der Name der Klasse, in der sich die statische Initialisierung befindet, hinzugefügt.

```
function translateStaticInit
    replace[repeat class_body_declaration]
        ClassBodyDecl[repeat class_body_declaration]
    by
        ClassBodyDecl[toStaticConstructor]
end function
% [2-1]change static initializer block to static constructor
rule toStaticConstructor
    replace[class_body_declaration]
        'static Block[block]
    % constructor name is same as class name
    import ClassName[class_name]
    construct NewStaticConstructor[constructor_declaration]
        'static ClassName()
          Block
    by
        NewStaticConstructor
end rule
```

Abbildung 31: TXL TranslateInitializers.Rul

6.1.5 TranslateMembers.Rul

Diese Regeldatei wird benutzt, um Konstruktoren, Methoden oder geschachtelte Deklarationen zu transformieren. Sie entfernt außerdem Modifizierer, für welche es in C#, in diesem Zusammenhang, keinen geeigneten Ersatz gibt. Beispielsweise *final*, *transient* oder *volatile*. Grundlage des TXL-Programmes waren die Java Version 1.1 und C# 3.0.

```
% [1-1-6]-------------------------------------------------------
% remove final,transient and volatile
% not exist in C#
rule removeNonCS
    replace [repeat modifier]
        CurrentModifier[modifier]
        RemainingModifiers[repeat modifier]
    where CurrentModifier[isFinal][isTransient][isVolatile]
    by
        RemainingModifiers
end rule
```

Abbildung 32. TXL TranslateMembers.Rul

6.1.6 TranslateFieldDeclaration.Rul

Felddeklarationen werden mit den Funktionen dieser Regeldatei transformiert. Beispielsweise folgt die Dimension eines Feldes in Java dem Variablenname und nicht, wie in C#, dem Typ des Feldes. Modifizierer der Sprache Java, welche in C# nicht existieren oder in diesem Zusammenhang eine andere Bedeutung haben, werden, unter Verwendung dieser Funktionen, in äquivalente Modifizierer transformiert.

```
% [1-5]change a field declaration int b[] to int[]b ----------------------------
function changeFieldArrayDimensions
    replace[field_declaration]
        Modifiers[repeat modifier]
        DataType[type_name]
        VarName[declared_name]
        Dim[repeat dimension]
        VarInit[opt equals_variable_initializer];
    % data type first and [] follows
    construct NewTypeSpecifier[type_specifier]
        DataType Dim
    by
        Modifiers NewTypeSpecifier VarName VarInit ;
end function
```

Abbildung 33: TXL TranslateFieldDeclaration.Rul

6.1.7 TranslateBlockStatements.Rul

Die Funktionen und Regel dieser Datei haben die Aufgabe Blöcke einer Methode oder eines Konstruktors zu transformieren. Anweisungen, Variablen- oder Konstantendeklarationen und Kontrollstrukturen, welche ein Block beinhalten kann, werden, wenn dies möglich ist, ebenfalls transformiert.

```
% [0]find a block as the body of a method or constructor or a statementt
function translateBlock
    replace * [block]
        '{
            DeclOrStmt[repeat declaration_or_statement]
        '}
    by
        '{
            DeclOrStmt[translateVarDeclaration]
                      [translateClassInBlock]
                      [translateStatementInBlock]
        '}
end function
```

Abbildung 34: TXL TranslateBlockStatements.Rul

6.1.8 HelperRules.Rul

Diese Datei stellt allgemeine Funktion zur Verfügung, welche von allen Regeldateien zur Transformation genutzt werden. Diese Funktionenkonnten keiner speziellen Regeldatei zugeordnet werden.

```
10  % [1] first find a java primitive
11  % then change it
12  function changeDataTypes
13      replace[type_specifier]
14          DataType[primitive_type]
15      by
16          DataType[changePrimDataTypes]
17  end function
18  % [2]To change the primitive data types
19  % byte to sbyte,boolean to bool
20  % other data types are the same
21  function changePrimDataTypes
22      replace [primitive_type]
23          JavaType[primitive_type]
24          import PrimDataTypesMapping [DataMapper]
25          deconstruct * [primitiveTable] PrimDataTypesMapping
26              JavaType -> CSType [primitive_type]
27      by
28          CSType
29  end function
```

Abbildung 35. TXL HelperRules.Rul

6.1.9 JavaToC#.Txl

In der TXL-Programmdatei werden alle Grammatik- und Regeldateien eingebunden. Sie beinhaltet die Funktion mit dem Nichtterminal *main*, welche die erste Funktion ist, die bei einer Transformation aufgerufen wird.

```
6   % The base Java grammar and grammar overrides
7   include "Java.Grm"
8   include "JavaCommentOverrides.Grm"
9   % Translation rule sets
10  include "DataStructures.Grm"
11  include "HelperRules.Rul"
12  include "TranslateMembers.Rul"
13  include "TranslateInitializers.Rul"
14  include "TranslateFieldDeclaration.Rul"
15  include "translateBlockStatements.Rul"
16
17  % Main translation rule
18  function main
19      replace [program]
20          PHeader[opt package_header]
21          ImportDeclaration[repeat import_declaration]
22          TypeDeclr[repeat type_declaration]
```

Abbildung 36: TXL JavaToC#.Txl

6.2 Transformation anhand eines Beispiels

Dieses Unterkapitel erklärt, wie eine Transformation mit der TXL durchgeführt wird. Es wird gezeigt, welche Funktionen und Regeln benutzt werden und wie sie dabei den Syntaxbaum transformieren. Es werden ausschließlich Funktionen und Regeln erklärt, welche bei der Transformation zur Anwendung kommen.

```
package examples;
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}
```

```
using System;
namespace examples
{
    class HelloWorld
    {
        public static void Main (String [] args)
        {
            Console.WriteLine ("Hello World");
        }
    }
}
```

Abbildung 37: Quelltext Java und C#

Die Abbildung 37 zeigt Konsolenanwendungen in den Sprachen Java und C#. Dabei ist in der rechten Abbildung das Resultat einer Transformation mit dem TXL-Programm *JavaToC#* zu sehen. Im Folgenden wird erklärt, wie dies erreicht wird.

Zunächst führt das TXL-Programm eine Analyse des Quelltextes durch. Dies geschieht unter Verwendung der Grammatikdateien und beginnt an der Definition des Nichtterminalsymboles *program*.

```
68    define program
69        [package_declaration]
70    end define

75    define package_declaration
76        [opt package_header]
77        [repeat import_declaration]
78        [repeat type_declaration]
79    end define

81    define package_header
82        'package [package_name] '; [NL][NL]
83    end define

85    define package_name
86         [qualified_name]
87    end define

691   define qualified_name
692       [reference]
693   end define

695   define reference
696       [id] [repeat component]
697   end define
```

Abbildung 38: TXL Java.Grm

Dabei wird, unter Verwendung der Grammatikdateien, rekursiv untersucht, in welche syntaktischen Einheiten der gelesene Token-Strom, bei einer Übereinstimmung, zu unterteilen ist. Im Anschluss an diese Analyse stellt sich die interne Struktur wie in Abbildung 39 dar. Sie zeigt den Syntaxbaum der Konsolenanwendung aus Abbildung 37. Dieser Syntaxbaum bildet die Grundlage für die darauf folgenden Transformationen.

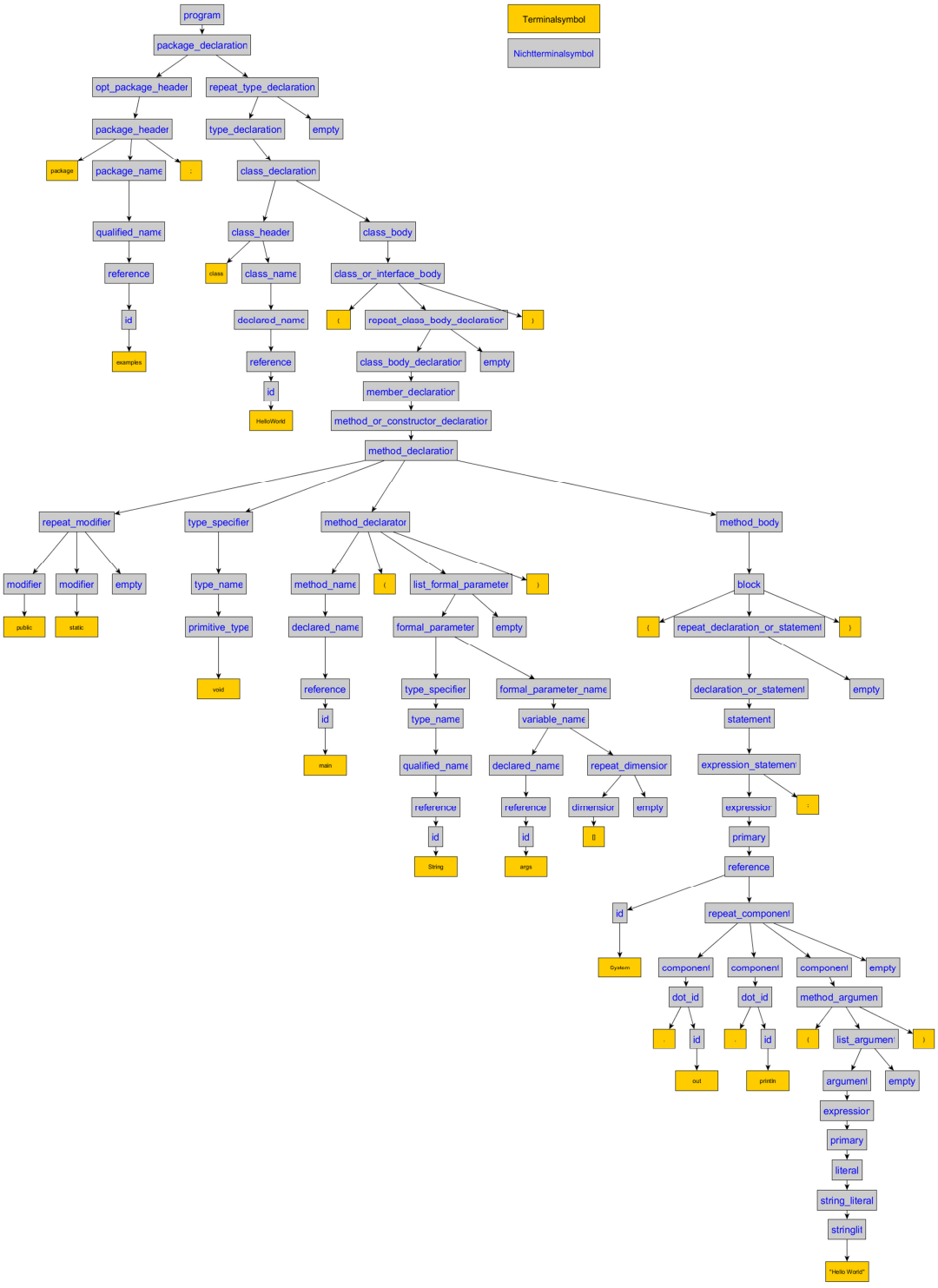

Abbildung 39: TXL Syntaxbaum

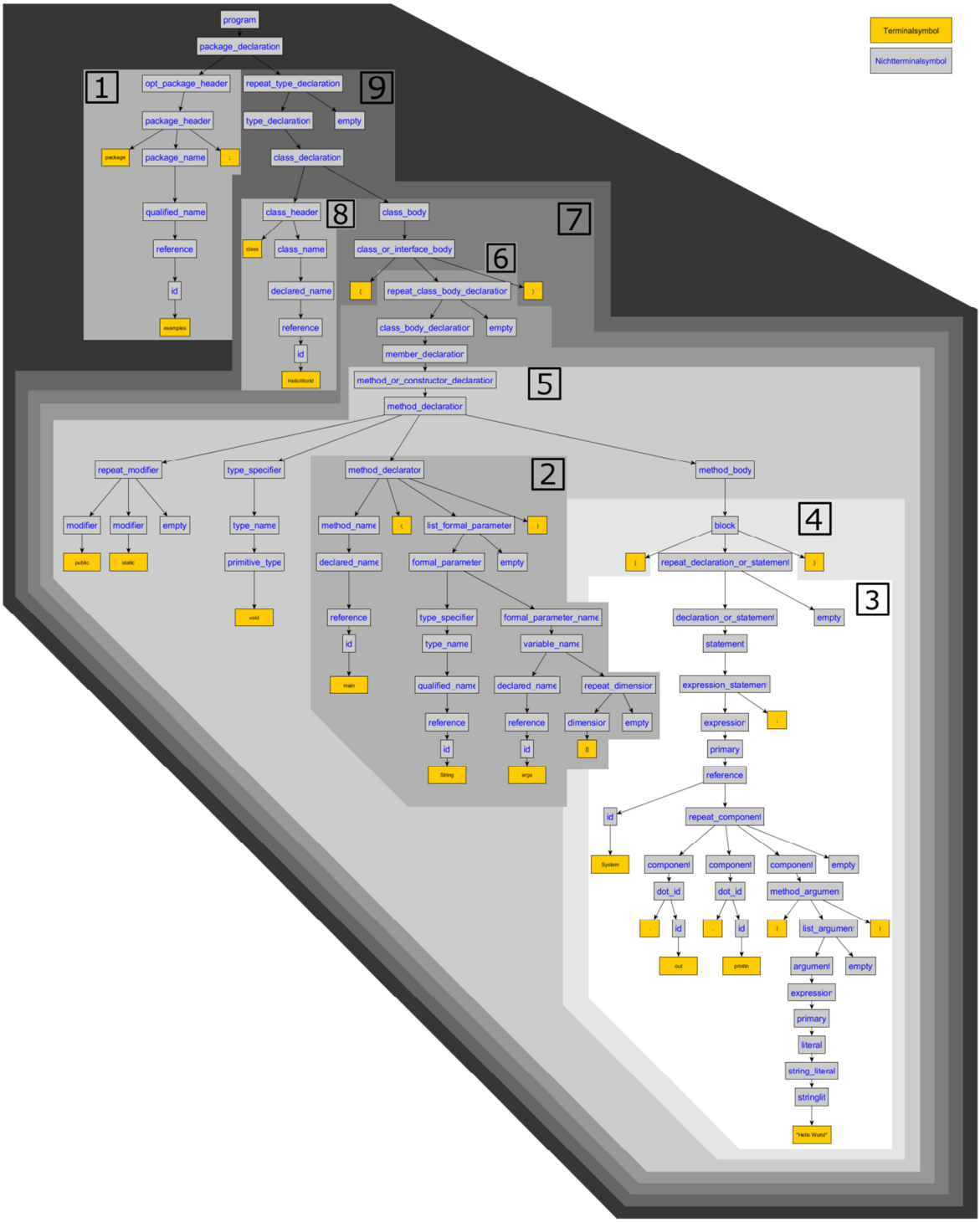

Abbildung 40: TXL Syntaxbaum in Teilbäume unterteilt

Im Anschluss an diese Analyse-Phase wird die Funktion *main*, welche sich in der Datei *JavaToC#.Txl* befindet, aufgerufen.

```
17  % Main translation rule
18  function main
19      replace [program]
20          PHeader[opt package_header]
21          ImportDeclaration[repeat import_declaration]
22          TypeDeclr[repeat type_declaration]
```

Abbildung 41: TXL Funktion main

In Zeile 19 wird das Nichtterminal, also die Wurzel des Syntaxbaumes, welcher transformiert werden soll, festgelegt. Dies bedeutet, dass das nachfolgende Muster, welches in den Zeilen 20 bis 22 angegeben ist, unter dem angegebenen Wurzelelement existieren muss. Wie in Abbildung 41 zu sehen, stimmt das Muster mit dem Syntaxbaum überein. Auf jeden erkannten Teilbaum verweist eine Variable, welche ebenfalls in Zeile 20 bis 22 angelegt wurden. Dadurch wird es möglich, dass nur einzelne Teilbäume, während der Transformation, bearbeitet werden. Diese lassen sich dann über ihre Variablennamen ansprechen.

Während der Transformation wird der Syntaxbaum in mehrere Teilbäume unterteilt. Abbildung 40 zeigt die vollständige Unterteilung des Syntaxbaumes. Die Variable *PHeader* verweist auf den Teilbaum mit der Nummer 1 und *TypeDeclr* auf die Nummer 9.

```
66          construct NewProgram[program]
67              NewImportDeclaration
68              PHeader[changePackageToNamespace]
69              TypeDeclr[changeClassHeader][changeInterfaceHeader]
```

Abbildung 42: TXL Unterteilung Syntaxbaum

Mittels des Konstruktors, welcher sich in der *main* Funktion befindet, wird ein neuer Teilbaum erstellt. Dieser hat als Wurzelelement das Nichtterminal *program*, welches in der Java.Grm (Abb. 38) definiert wurde. Durch ein *redefine* des Nichtterminals *package_declaration* (Abb. 43), ist das Ersetzungsmuster des Konstruktors syntaktisch korrekt. Der neu erzeugte Baum (Abb. 44) besteht aus den Nichtterminalsymbolen *repeat_import_declaration* und *namespace_member_declaration*.

```
103 redefine package_declaration
104         ...%Java
105       |
106         [repeat import_declaration]%C#
107         [namespace_member_declaration]
108 end redefine
```

```
94  define namespace_member_declaration
95      [opt package_header] [IN]
96      [opt '{][NL][IN]
97          [repeat type_declaration][EX]
98      [opt '}][NL][IN]
99  end define
```

Abbildung 43: TXL redefine

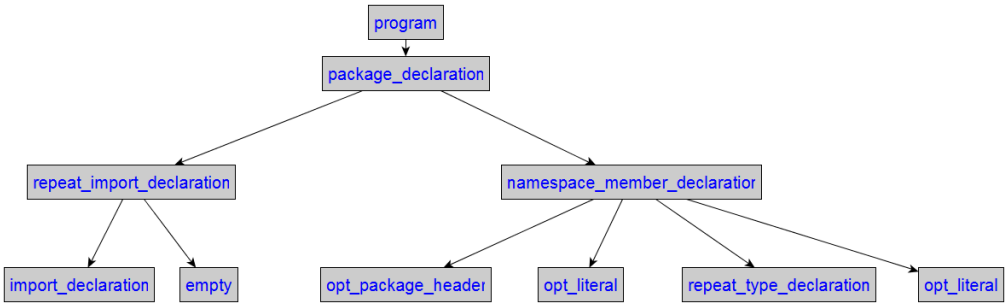

Abbildung 44: TXL Syntaxbaum - Neue Struktur

Auf die in dem Konstruktor angegebenen Variablen, welche auf die Teilbäume referenzieren, werden nun die Funktionen angewendet. Zunächst wird die Funktion *changePackageToNamespace* auf den Teilbaum mit der Nummer 1 ausgeführt.

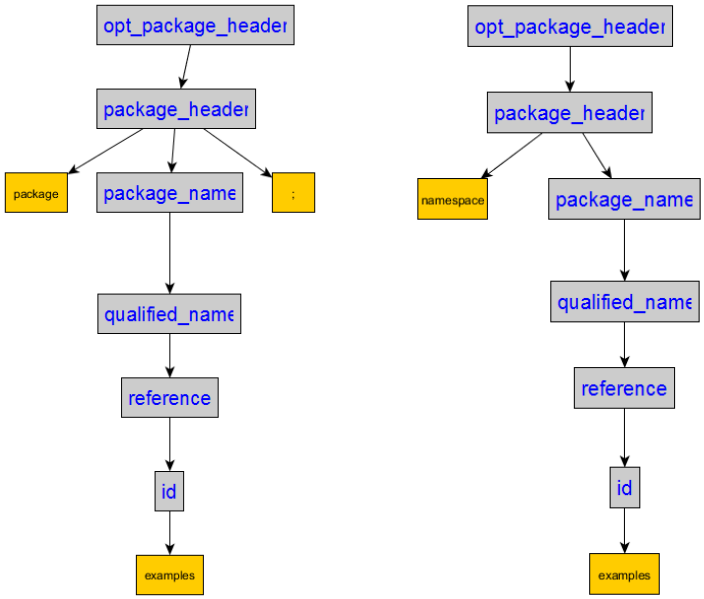

Abbildung 45: TXL Teilbaum vor und nach der Transformation

```
146  function changePackageToNamespace
147      replace [opt package_header]
148          'package Name[package_name] ';
149      by
150          'namespace Name
151  end function
```

Abbildung 46: TXL Funktion changePackageToNamespace

Die Funktion überprüft den Teilbaum auf das angegebene Muster in Zeile 148. Dabei kommt es zu einer Übereinstimmung. Die Variable Name referenziert auf den Teilbaum *package_name*. Dieser wird in dem Ersetzungsmuster übernommen. Die Abbildung 45

zeigt den Teilbaum vor und nach dieser Transformation. Damit ist die Transformation des Teibaumes abgeschlossen.

Als nächstes wird die Funktion *changeClassHeader* auf den Teilbaum mit der Nummer 9 angewendet. Dabei wird dieser wiederum in Teilbäume unterteilt. *ClassHead* verweist auf den Teilbaum mit der Nummer 8, *ClassBody* auf die Nummer 7 und *Remaining* auf weitere *type_declaration*.

```
160  function changeClassHeader
161      replace [repeat type_declaration]
162          ClassHead[class_header]ClassBody[class_body]
163          Remaining [repeat type_declaration]
164      by
165          NewClassHead[addClassExtendToImplmt]
166          ClassBody [translateEmptyBody][changeClassBody]
167          Remaining[changeClassHeader][changeInterfaceHeader]
168  end function
```

Abbildung 47: TXL Funktion changeClassHeader

Im Ersetzungsmuster wird auf den Teilbaum mit der Nummer 7 die Funktion *changeClassBody* angewendet. Diese Funktion nutzt wiederum eine Variable im Suchmuster, welche den Teilbaum in einen weiteren Teilbaum unterteilt. Wie bereits erwähnt, wird während der Transformation der Syntaxbaum in mehrere Teilbäume unterteilt, welche einzeln auf Suchmuster untersucht und bei einer Übereinstimmung transformiert werden. Wie in Abbildung 40 zu sehen, wird der Syntaxbaum in neun Teilbäume unterteilt. Im Folgenden soll erklärt werden, welche Funktionen oder Regeln die Teilbäume bearbeiten. Prinzipiell werden alle Funktionen des TXL-Programmes angewendet. Kommt es dabei aber nicht zu einer Transformation, bleiben die Teilbäume unverändert. Diesbezüglich werden nur die Funktionen und Regeln näher erklärt, welche auch tatsächlich eine Transformation der Teilbäume durchführen.

6.2.1 Transformation des Teilbaumes 2

Die Funktion *changeMethodDeclarator* ist eine Funktion, welche auf den Teilbaum mit der Nummer 2 angewendet wird.

```
51  function changeMethodDeclarator
52      replace [method_declarator]
53          Name[method_name] '( FormalParms[list formal_parameter] ') Dim[repeat dimension]
54          construct NewFormalParms[list formal_parameter]
55           _[changeFormalParamsDataTypes each FormalParms ]
56
57      by
58          Name[changeMain] '( NewFormalParms ')Dim
59  end function
```

Abbildung 48. TXL Funktion changeMethodDeclarator

Zunächst wird der Teilbaum auf das vorgegebene Muster untersucht. Den Variablen werden die Teilbäume method_name, list formal_parameter und Dim zugewiesen. Während der Ersetzung wird auf die Variable *Name* die Funktion *changeMain* angewendet. Sie hat die Aufgabe, den Methodenname *main* in *Main* zu tranformieren.

```
205  function changeMain
206  replace[method_name]
207      'main
208  by
209      'Main
210  end function
```

Abbildung 49: TXL Funktion changeMain

Die Funktion *changeMethodConArrayDimentions* wird ebenfalls auf diesen Teilbaum angewendet. Sie hat die Aufgabe, die Liste von Formalen Parametern, auf welche sie angewendet wird, zu zerlegen und bei der Ersetzung die Variable *Dim* mit der Variable *Name* zu tauschen.

```
232  rule changeMethodConArrayDimentions
233      replace[formal_parameter]
234          FormalParm[formal_parameter]
235      deconstruct FormalParm
236          FinalOpt[opt 'final] DataType[type_name] Name[declared_name]Dim[repeat dimension]
237      by
238          FinalOpt DataType Dim Name
```

Abbildung 50: TXL Funktion changeMethodConArrayDimentions

Die Abbildung 51 zeigt den Teilbaum vor und nach der Transformation.

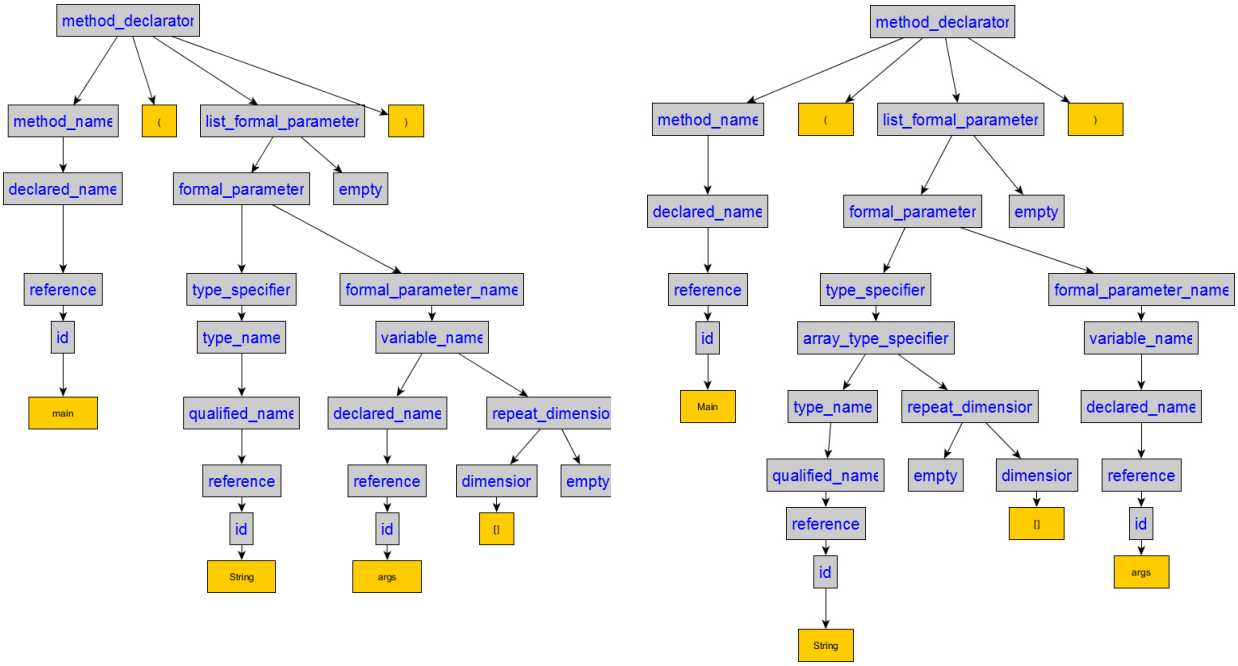

Abbildung 51: TXL Teilbaum vor und nach der Transformation

6.2.2 Transformation des Teilbaumes 3

Die Funktion *changeCSStatement* transformiert beispielsweise die Anweisung *System.out.println* in die äquivalente Anweisung *Console.WriteLine*. Dazu wird das Suchmuster vom Typ *expression* im dritten Teilbaum überprüft und den Variablen zugewiesen. Über den Konstrukter *Key* wird ein Teilbaum vom Typ *reference* erzeugt und mit Referenzen der Variablen belegt.

```
400  function changeCSStatement
401      replace [expression]
402          FirstId[id] '. SecondId[id]'. ThirdId[id]Remaining[repeat component]
403      %to search for specified entry in the statemnts table
404      construct Key[reference]
405          FirstId.SecondId.ThirdId
406      import StatementMapping [StmtMapper]
407      % match the key entry with C# one
408      deconstruct *[table_entry] StatementMapping
409          Key -> CSStmt [reference]
410      % to build a new valid expression
411      deconstruct CSStmt
412          FirstCSId[id] CSRemaining[repeat component]
413      %join the method arguements with the new part
414      by
415          FirstCSId CSRemaining[. Remaining]
416  end function
```

Abbildung 52: TXL Funktion changeCSStatement

Der Dekonstrukter, vom Typ *table_entry*, zerlegt die Tabelleneinträge aus der Import-Anweisung, welche durch eine Export-Anweisung, in der Funktion *main*, erstellt wurden. Dabei werden nur die Einträge beachtet, welche mit dem Teilbaum *Key* übereinstimmen. Ein weiterer Dekonstrukter, mit dem Name *CSStmt*, zerlegt die Variable *CSStmt* aus dem vorherigen Dekonstruktor und weist sie den Variablen *FirstCSId* und *CSRemaining* zu.

```
33    export StatementMapping [StmtMapper]
34    'System.out.println -> 'Console.WriteLine
35    'System.out.print -> 'Console.Write
```

Abbildung 53: TXL export StatementMapping

Das Ersetzungsmuster setzt sich demzufolge aus der Variablen *FirstCSId* und einer Folge der Variablen *CSRemaining*. *CSRemaining* setzt sich zusammen aus *CSRemaining* und alles folgenden *repeat component*.

Die Abbildung 54 zeigt den Teilbaum vor und nach der Transformation.

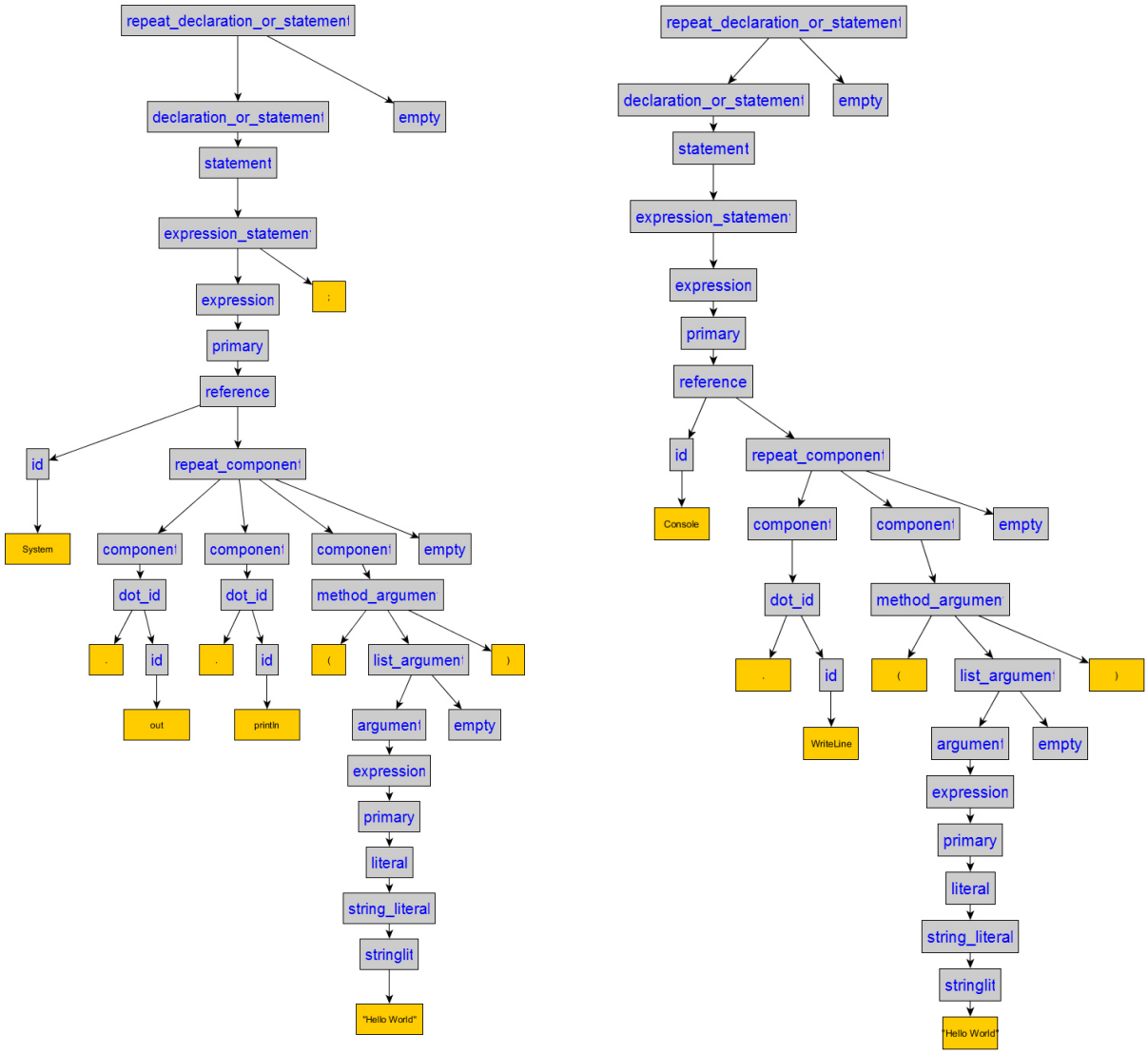

Abbildung 54: TXL Teilbaum vor und nach der Transformation

7 Zusammenfassung und Ausblick

In diesem Kapitel wird die vorliegende Arbeit noch einmal zusammengefasst und ausgewertet. Das Ziel der Arbeit war die Untersuchung der Möglichkeiten der automatisierten Erstellung eines Compilers zur Transformation von Quellcode aktueller Programmiersprachen untereinander.

Zunächst wurde in Kapitel 2 eine theoretische Einführung zum Thema Compilerbau gegeben. Dieses Kapitel diente dem besseren Verständnis der folgenden Kapitel. Es zeigte, aus welchen Bestandteilen ein Compiler besteht und welche Phasen dabei bearbeitet werden.

Das Kapitel 3 untersuchte die Möglichkeiten der automatisierten Erstellung einzelner Compiler-Komponenten. Dabei wurde erkannt, dass die automatisierte Erstellung eines Parsers in Verbindung mit einem Scanner zwar die Analyse-Phase eines Compilers in vollem Umfang abdeckt, aber die Transformation des erzeugten Syntaxbaumes nicht mit diesen Komponenten möglich ist. Darüber hinaus wurde die Möglichkeit der automatisierten Erstellung eines Transcompilers untersucht.

Das Kapitel 4 untersuchte die Turing eXtender Language. Es wurde der geschichtliche Hintergrund erklärt und es wurde erkannt, dass die TXL für eine Transformation von Quellcode bestens geeignet ist.

Aufgrund der Ergebnisse aus Kapitel 4 wurden in Kapitel 5 die Grundlagen zum Erstellen eines TXL-Programmes näher untersucht und ausgeführt.

Das Kapitel 6 zeigte, wie eine Transformation mit der TXL durchgeführt wird. Dieses Kapitel hat bewiesen, dass eine automatisierte Erstellung eines Compilers, zur Transformation von Quellcode, möglich ist.

Nach der Bearbeitung dieses Themas in Verbindung mit der TXL wurde erkannt, wie umfangreich eine Erstellung eines TXL-Programmes ist. Dennoch lohnt es sich mit der TXL einen Transcompiler für die Transformation zu erstellen. Der Aufwand ist sicherlich nicht zu unterschätzen, aber ein fertiges Programm, zur Transformation, ist von unschätzbarem Wert. Probleme, welche noch weiter untersucht werden müssen, sind beispielsweise die Übersetzung ganzer Programme, die Übersetzung der Sprachspezifischen API. Für jedes Konstrukt der eine Sprache muss ein äquivalentes Konstrukt in der anderen Sprache gefunden werden

Nachfolgende Arbeit, für eine einfachere Programmierung eines TXL-Programmes, könnte die Programmierung eines Tools sein, welches eine Grammatikbeschreibung, aufgrund der Analyse einer Quellsprache, automatisiert erstellt.

Literatur und verwendete Tools/Programme

[TXL13] Offizielle Webseite,
http://txl.ca/index.html, verfügbar 05/2014,
Version: 07/2013

[CDMS02] CORDY, James R.; DEAN, Thomas R.; MALTON, Andrew J.; SCHNEIDER, Kevin A.: Source transformation in software engineering using the TXL transformation system. In: *Information & Software Technology* 44 (2002), Nr. 13, S. 827-837

[ETC11] CORDY, James R.: Excerpts from the TXL Cookbook,
In: *Generative and Transformational Techniques in Software Engineering*, (01/2011), S.27-91

[EEA05] EL-RAMLY, Mohammad; ELTAYEB, Rihab; ALLA, Hisham A.:
An Experiment in Automatic Conversion of Legacy
Java Programs to C#, (2005)

[TPL12] CORDY, James R.; CARMICHAEL, Ian H.; HALLIDAY, Russell:
The TXL Programming Language, (07/2012), Version10.6

[UGTCI12] CORDY, James R.: User´s Guide to the TXL Compiler / Interpreter (07/2012), Version 10.6

[MSDNM] Offizielle Webseite - MSDN Magazin
Erstellen eines Sprachcompilers für .NET Framework,
http://msdn.microsoft.com/de-de/magazine/cc136756.aspx#S3
verfügbar am 05/2014

[KP13] KÖHLER, P.: Compilerbau-Praktikum, (2013)

[CI02] Skript über Compiler und Interpreter
http://www.iai.uni-bonn.de/III/lehre/vorlesungen/Informatik_II/SS02/folien/B9sw4.pdf
verfügbar am 05/2014

[AR] Aufbau und Arbeitsweise von Compilern
http://www2.math.uni-wuppertal.de/~axel/skripte/compiler/c1_01.html
verfügbar am 05/2014

[WC14] Wikipedia; Compiler
http://de.wikipedia.org/wiki/Compiler
verfügbar am 05/2014

[WT14] Wikipedia; Token (Übersetzerbau)
http://de.wikipedia.org/wiki/Token_(%C3%9Cbersetzerbau)
verfügbar am 05/2014

[WS14] Wikipedia; Syntaxbaum
http://de.wikipedia.org/wiki/Syntaxbaum
verfügbar am 05/2014

[WTOK14] Wikipedia; Tokenizer
http://de.wikipedia.org/wiki/Lexikalischer_Scanner
verfügbar am 05/2014

[WCB14] Wikipedia; Compilerbau
http://de.wikipedia.org/wiki/Compilerbau
verfügbar am 05/2014

[WPG14] Wikipedia; Parsergenerator
http://de.wikipedia.org/wiki/Parsergenerator
verfügbar am 05/2014

[WP14] Wikipedia; Parser
http://de.wikipedia.org/wiki/Parser
verfügbar am 05/2014

[WLEX14] Wikipedia; Lex
http://de.wikipedia.org/wiki/Lex_(Informatik)
verfügbar am 05/2014

[WYACC14] Wikipedia; Yacc
http://de.wikipedia.org/wiki/Yacc
verfügbar am 05/2014

[WGB14] Wikipedia; GNU Bison
http://de.wikipedia.org/wiki/GNU_Bison
verfügbar am 05/2014

[GNU14] Webseite; GNU
http://www.gnu.org/software/bison
verfügbar am 05/2014

[WEBNF14] Wikipedia; Erweiterte Backus-Naur-Form
http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form
verfügbar am 05/2014

[WKG14] Wikipedia; Kontextfreie Grammatik
http://de.wikipedia.org/wiki/Kontextfreie_Grammatik
verfügbar am 05/2014

[WRA14] Wikipedia; Reguläre Ausdrücke
http://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck
verfügbar am 05/2014

[CB14] Webseite; Compilerbau
http://www2.in.tum.de/hp/Main?nid=79
verfügbar am 05/2014

Anlagen

A1, Java.Grm

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL Java 1.1 Basis Grammar
% Copyright 2001 TXL Software Research Inc. and James R. Cordy

% Adapted from the Java Language Specification, 2nd edition

% Modified by Xinping Guo May 28, 2002
% -- Defined a new nonterminal type called "declared_name".
% -- "class_name", "interface_name", "variable_name" and "method_name" are "declared_name".
% -- Changed "reference" to "variable_name" in "catch_clause".
% -- In "constructor_declarator", changed "type_name" to "class_name".
% -- In "implements_clause", changed "class_name+" to "qualified_name+".
% -- Defined a new nonterminal type called "method_or_constructor_declaration".
% -- In "member_declaration", removed "field_declaration" and changed
%        "method_declaration" to "method_or_constructor_declaration".
% -- In "class_body_declaration", removed "constructor_declaration" and
%        added "field_declaration".

% Modified by Xinping Guo June 12, 2002
% The changes made expansion of dot notation much easier.
% -- Redefined "primary". Removed "primary_component" and "dot_primary_component"
% -- Redefined "subprimary".
% -- Removed "field_access", "method_invocation" and "constructor_invocation".
% -- Redefined "reference".
% -- Defined "component".
% -- Defined "method_argument".

% Modified by Xinping Guo July 17, 2002
% -- Added [repeat component] after "( [expression] ') in defintion of primary.

% Modified by Xinping Guo July 23, 2002
% -- Added 'strictfp in defintion of modifier.



% Lexical conventions of Java

#pragma -width 128

tokens
    number    "\d+[LlFfDd]?"                                    % Java ints
              + "[(\d+(.\d*))(.\d+)]([eE][+-]?\d+)?[LlFfDd]?"   % Java float constants
              + "0[Xx][\dABCDEFabcdef]+[LlFfDd]?"               % Java hex constants
end tokens

comments
    //
    /*  */
end comments

compounds
    !=  '% =  &&  &=  *=  ++  +=   --  -=  /=  ^=  |=  ||
    <<  <<=  <=  ==  >=  >>  >>=  >>>  >>>=
end compounds


% Removed this and super - XG June 19, 2002
keys
    'abstract 'boolean 'break 'byte 'case 'cast 'catch 'char 'class 'const 'continue
    'default 'do 'double 'else 'extends 'false 'final 'finally 'float 'for 'future
    'goto 'if 'implements 'import 'inner 'instanceof 'int 'interface 'long
    'native 'new 'null 'package 'private 'protected 'public 'return
    'short 'static  'switch 'synchronized
    'throw 'throws 'transient 'true 'try 'void 'volatile 'while
end keys


% Syntax of Java 1.1

define program
    [package_declaration]
end define


% Declarations

define package_declaration
    [opt package_header]
    [repeat import_declaration]
    [repeat type_declaration]
end define

define package_header
    'package [package_name] '; [NL][NL]
end define

define package_name
    [qualified_name]
end define

define import_declaration
    'import [imported_name] '; [NL][NL]
end define

define imported_name
    [package_or_type_name] [opt dot_star]
end define

define package_or_type_name
    [qualified_name]
end define

define dot_star
    '. '*
end define

define declared_name
   [reference]
end define

% Class and interface declarations

define type_declaration
        [class_declaration]      [NL][NL]
    |   [interface_declaration]  [NL][NL]
end define

define class_declaration
    [class_header] [class_body]
end define

define class_header
    [repeat modifier] 'class [class_name] [opt extends_clause] [opt implements_clause]
end define

define class_body
    [class_or_interface_body]
end define

define interface_declaration
    [interface_header] [interface_body]
end define

define interface_header
    [repeat modifier] 'interface [interface_name] [opt extends_clause] [opt
    implements_clause]
end define

define interface_body
    [class_or_interface_body]
end define

define modifier
        'abstract
    |   'final
    |   'public
    |   'protected
    |   'private
    |   'static
    |   'transient
    |   'volatile
    |   'native
    |   'synchronized
    |   'strictfp       % Added strictfp XG July 23, 2002
end define

define extends_clause
    'extends [list type_name+]
end define

define implements_clause
    'implements [list qualified_name+]
end define

define class_name
    [declared_name]
end define

define interface_name
    [declared_name]
end define

define class_or_interface_body
    '{                                          [NL][IN]
       [repeat class_body_declaration]          [EX]
    '} [opt ';]                                 [NL][NL]
end define

define class_body_declaration
        [empty_declaration]
    |   [member_declaration]
    |   [instance_initializer]
    |   [static_initializer]
    |   [field_declaration]
end define

define empty_declaration
    '; [NL]
    |'//
end define

define member_declaration
       [method_or_constructor_declaration]
    |  [type_declaration]     % [class_declaration] or [interface_declaration]
end define

define method_or_constructor_declaration
        [method_declaration]
    |   [constructor_declaration]
end define

define instance_initializer
    [NL] [block]              [NL][NL]
end define

define static_initializer
    [NL] 'static [block]      [NL][NL]
end define

define constructor_declaration
    [NL] [repeat modifier] [constructor_declarator] [opt throws] [constructor_body] [
    NL][NL]
end define

define constructor_declarator
    [class_name] '( [list formal_parameter] ')
end define

define constructor_body
    [block] [NL]
end define


% Field declarations

define field_declaration
    [variable_declaration]
end define

define variable_declaration
    [repeat modifier] [type_specifier] [variable_declarators] '; [NL]
end define

define variable_declarators
    [list variable_declarator+]
end define

define variable_declarator
```

```
    [variable_name] [opt equals_variable_initializer]
end define

define variable_name
    [declared_name][repeat dimension]
end define

define equals_variable_initializer
    '= [variable_initializer]
end define

define variable_initializer
        [expression]
    |   [array_initializer]
end define

define array_initializer
    '{ [list variable_initializer] [opt ',] '}
end define


% Method declarations

define method_declaration
    [NL] [repeat modifier] [type_specifier] [method_declarator] [opt throws] [
    method_body]
end define

define method_declarator
    [method_name] '( [list formal_parameter] ') [repeat dimension]
end define

define method_name
    [declared_name]
end define

define formal_parameter
    [opt 'final] [type_specifier] [formal_parameter_name]
end define

define formal_parameter_name
    [variable_name]
end define

define throws
    'throws [list qualified_name+]
end define

define method_body
        [block]         [NL][NL]
    |   ';              [NL][NL]
end define


% Type specifiers

define type_specifier
        [type_name]
    |    [array_type_specifier]


end define

define array_type_specifier
    [type_name] [repeat dimension+]
end define

define type_name
        [primitive_type]
    |   [qualified_name]
end define

define primitive_type
        'boolean
    |   'char
    |   'byte
    |   'short
    |   'int
    |   'long
    |   'float
    |   'double
    |   'void
end define


% Statements

define block
    '{                                          [NL][IN]
        [repeat declaration_or_statement]       [EX]
    '}
end define

define declaration_or_statement
        [local_variable_declaration]
    |   [class_declaration]
    |   [statement]
end define

define local_variable_declaration
    [variable_declaration]
end define

define statement
        [label_statement]
    |   [empty_statement]
    |   [expression_statement]
    |   [if_statement]
    |   [switch_statement]
    |   [while_statement]
    |   [do_statement]
    |   [for_statement]
    |   [break_statement]
    |   [continue_statement]
    |   [return_statement]
    |   [throw_statement]
    |   [synchronized_statement]
    |   [try_statement]
    |   [block]
end define


define empty_statement
    ';                      [NL]
end define

define label_statement
    [label_name] ':         [NL]
end define

define label_name
    [reference]
end define

define expression_statement
    [expression] ';         [NL]
end define

define if_statement
    'if '( [expression] ')
        [statement]
    [opt else_clause]       [NL]
end define

define else_clause
    'else
        [statement]
end define

define switch_statement
    'switch '( [expression] ') [switch_block]  [NL]
end define

define switch_block
    '{                                          [IN][NL]
        [repeat switch_alternative]             [EX]
    '}
end define

define switch_alternative
    [switch_label]                              [IN][NL]
        [repeat declaration_or_statement]       [EX]
end define

define switch_label
        'case [constant_expression] ':
    |   'default ':
end define

define while_statement
    'while '( [expression] ')
        [statement]                             [NL]
end define

define do_statement
    'do
        [statement]
    'while '( [expression] ') ';                [NL]
end define


define for_statement
    'for '( [for_init] [for_expression] [for_update] ')
        [statement]                             [NL]
end define

define for_init
        [list expression] ';
    |   [local_variable_declaration]
end define

define for_expression
    [opt expression] ';
end define

define for_update
    [list expression]
end define

define break_statement
    'break [opt label_name] ';        [NL]
end define

define continue_statement
    'continue [opt label_name] ';     [NL]
end define

define return_statement
    'return [opt expression] ';       [NL]
end define

define throw_statement
    'throw [expression] ';            [NL]
end define

define synchronized_statement
    'synchronized '( [expression] ')
        [statement]                   [NL]
end define

define try_statement
    'try [block] [repeat catch_clause] [opt finally_clause]     [NL]
end define

define catch_clause
%   'catch '( [type_specifier] [reference] ') [block]
    'catch '( [type_specifier] [variable_name] ') [block]  % July 15
end define

define finally_clause
    'finally [block]
end define


% Expressions

define constant_expression
    [expression]    % which is a constant
end define
```

```
define expression
    [assignment_expression]
end define

define assignment_expression
        [conditional_expression]
    |   [unary_expression] [assignment_operator] [assignment_expression]
end define

define assignment_operator
        '=
    |   '*=
    |   '/=
    |   '% =
    |   '+=
    |   '-=
    |   '<<=
    |   '>>=
    |   '>>>=
    |   '&=
    |   '^=
    |   '|=
end define

define conditional_expression
    [conditional_or_expression] [opt conditional_choice]
end define

define conditional_choice
    '? [expression] ': [conditional_expression]
end define

define conditional_or_expression
    [conditional_and_expression] [repeat or_conditional_and_expression]
end define

define or_conditional_and_expression
    '|| [conditional_and_expression]
end define

define conditional_and_expression
    [inclusive_or_expression] [repeat and_inclusive_or_expression]
end define

define and_inclusive_or_expression
    '&& [inclusive_or_expression]
end define

define inclusive_or_expression
    [exclusive_or_expression] [repeat or_exclusive_or_expression]
end define

define or_exclusive_or_expression
    '| [exclusive_or_expression]
end define

define exclusive_or_expression
    [and_expression] [repeat or_and_expression]
end define

define or_and_expression
    '^ [and_expression]
end define

define and_expression
    [equality_expression] [repeat and_equality_expression]
end define

define and_equality_expression
    '& [equality_expression]
end define

define equality_expression
    [relational_expression] [repeat equality_op_relational_expression]
end define

define equality_op_relational_expression
    [equality_op] [relational_expression]
end define

define equality_op
    '== | '!=
end define

define relational_expression
    [shift_expression] [repeat relational_op_shift_expression]
end define

define relational_op_shift_expression
        [relational_op] [shift_expression]
    |   'instanceof [type_specifier]
end define

define relational_op
    '<  | '> | '<= | '>=
end define

define shift_expression
    [additive_expression] [repeat shift_additive_expression]
end define

define shift_additive_expression
    [shift_op] [additive_expression]
end define

define shift_op
    '<< | '>> | '>>>
end define

define additive_expression
    [multiplicative_expression] [repeat add_op_multiplicative_expression]
end define

define add_op_multiplicative_expression
    [add_op] [multiplicative_expression]
end define

define add_op
    '+ | '-
end define

define multiplicative_expression
    [unary_expression] [repeat mult_op_unary_expression]
end define

define mult_op_unary_expression
    [mult_op] [unary_expression]
end define

define mult_op
    '* | '/ | '%
end define

define unary_expression
        [pre_inc_dec] [unary_expression]
    |   [unary_op] [unary_expression]
    |   [postfix_expression]
    |   [cast_expression]
end define

define pre_inc_dec
    '++ | '--
end define

define unary_op
    '+ | '- | '~ | '!
end define

define cast_expression
    '( [type_specifier] ') [unary_expression]
end define

define postfix_expression
    [primary] [repeat post_inc_dec]
end define

define post_inc_dec
    '++ | '--
end define

define subscript
    '[ [expression] ']
end define

define primary
        [literal]
    |   [reference]
    |   '( [expression] ') [repeat component]
    |   [class_instance_creation_expression]
    |   [array_creation_expression]
end define

define literal
        [numeric_literal]
    |   [character_literal]
    |   [string_literal]
    |   [boolean_literal]
    |   [null_literal]
end define

define numeric_literal
    [number]
end define

define character_literal
    [charlit]
end define

define string_literal
    [stringlit]
end define

define boolean_literal
        'true
    |   'false
end define

define null_literal
    'null
end define

define class_instance_creation_expression
    'new [class_or_interface_name] '( [list argument] ') [opt class_body]
end define

define class_or_interface_name
    [qualified_name]
end define

define array_creation_expression
    'new [array_type_specifier] [opt array_initializer]
end define

define dimension
    '[ [opt expression] ']
end define

define argument
    [expression]
end define

define qualified_name
    [reference]
end define

define reference
    [id] [repeat component]
end define

define component
      [dot_id]
   | [method_argument]
   | [subscript]
end define

define method_argument
```

```
    '( [list argument] ')
end define

define dot_id
    '. [id]
end define

% --- END ---

```

A2, DataStrucures.Grm

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL 103a3 Translation, Java to C#
% Rihab Eltayeb, Sudan University, August 2005
% [part of master thesis project]


define optRight-brace
    [opt '{]
end define
define optLeft-brace
    [opt '}]
end define
% for using in class and interface body section
% C# seperates the modifiers applied to classes from those to interfaces and methods
redefine modifier
    ... % java
    |[class_modifier]     % C#
    |[interface_modifier] % C#
    |[method_modifier]    % C#
    |[constant_modifier]  % C#
end redefine
% new C# set of modifiers
define interface_modifier
    'new
    |'public
    |'protected
    |'internal
    |'private
end define
% new C# set of modifiers
define class_modifier
    'new
    |'public
    |'protected
    |'internal
    |'private
    |'abstract
    |'sealed
end define
% new C# set of modifiers
define method_modifier
    'new
    |'public
    |'protected
    |'internal
    |'private
    |'static
    |'virtual
    |'override
    |'abstract
    |'extern
end define
% new C# set of modifiers for constants
define constant_modifier
    'const
    |'readonly
end define

% for using in data types section
redefine primitive_type
    ... % java
    |'sbyte % C#
        |'bool  % C#
end redefine
% for using in constructor section
define constructor_initializer
     ': 'base [repeat component]
    |': 'this [repeat component]
end define
% to add the inheritance syntax of C#
redefine constructor_declarator
    ... % java
    |[class_name] '( [list formal_parameter] ')[opt constructor_initializer]% C#
end define
% for using with a switch statement to prevent fall through
define goto_statement:
      'goto [reference][NL]
    | 'goto 'case [constant_expression]  [NL]
    | 'goto 'default  [NL]
    | 'goto [switch_label] [NL] % to allow putting the label and the : will be removed
end define
redefine statement
    ...
    |[goto_statement]
end redefine
% For synchronize statement
keys
...|
'lock
end keys
redefine synchronized_statement
    ...
    | 'lock '( [expression] ')
        [statement]              [NL]
end define
% The data structures
% contains an entry for every java modifier and the equivelant C# one.
define entries
    [modifier]
    |[primary]
    |[reference]
end define

define table_entry
    [entries] '-> [entries]
end define
define Mapper
    [repeat table_entry]
end define
define StmtMapper
    [repeat table_entry]
end define

% Data types
define primitive_entry
    [primitive_type]
end define
define primitiveTable
    [primitive_entry] '-> [primitive_entry]
end define
define DataMapper
    [repeat primitiveTable]
end define

% Exceptions
define exceptionEntries
    [qualified_name]
end define
define exceptionTable
    [exceptionEntries] '-> [exceptionEntries]
end define

define ExceptionMapper
    [repeat exceptionTable]
end define
```

A3, JavaCommentOverride.Grm

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% Overrides to preserve comments in Java
% Jim Cordy, March 2001 (Rev Feb 2003)

#pragma -comment

% Can have comments as statements
redefine statement
    ...
|   [comment_NL]
end redefine

% Can have comments as declarations
redefine class_body_declaration
    ...
|   [comment_NL]
end redefine

redefine package_declaration
    [repeat comment_NL] ...
end redefine

redefine import_declaration
    ...
|   [comment_NL]
end redefine

redefine type_declaration
    ...
|   [comment_NL]
end redefine

% Can have comments before initializers ...
redefine variable_initializer
    [repeat comment_NL] ...
end redefine

% ... or after initializers ...
redefine variable_initializer
    ... [repeat comment_NL]
end redefine

% ... or before =initializers
redefine equals_variable_initializer
    [repeat comment_NL] ...
end redefine

% Can have comments before formal parameters ...
redefine formal_parameter
    [repeat comment_NL] ...
end redefine

% ... or after formal parameters ...
redefine formal_parameter
    ...  [repeat comment_NL]
end redefine

% ... or after a parameter list
redefine constructor_declarator
    ... [repeat comment_NL]




end redefine

redefine method_declarator
    ... [repeat comment_NL]
end redefine

% Can have comments before arguments ...
redefine argument
    [repeat comment_NL] ...
end redefine

% ... or after arguments ...
redefine argument
    ...  [repeat comment_NL]
end redefine

% ... or after an argument list
redefine method_argument
    ... [repeat comment_NL]
end redefine

% Can have comments after an if condition
redefine if_statement
    'if '( [expression] ')  [repeat comment_NL]
        [statement]
    [opt else_clause]   [NL]
end redefine

% Can have comments on a block
redefine block
    [repeat comment_NL] ...
end redefine

% Can have comments before an expression
redefine expression
    [repeat comment_NL] ...
end redefine

% Need newlines after comments
define comment_NL
    [comment] [NL]
end define


```

A4, TranslateInitializers.Rul

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL 103a3
% Rihab Eltayeb, Sudan University, August 2005
% [part of master thesis project]
% % *****
% % common to instance initializer, static initializer
% ------------------------------------------------------------------------
%  ********* [1]INSTANCE INITIALIZER    *********
% ------------------------------------------------------------------------
% [0]find the initializers
function translateInstanceInit
    replace[repeat class_body_declaration]
       ClassBodyDecl[repeat class_body_declaration]
    where ClassBodyDecl[containInitBlock]
    % change initializers by methods
    construct NewClassBodyDecl[repeat class_body_declaration]
        ClassBodyDecl[toMethods]

    by
    % add all the methods calls in all constructors
        NewClassBodyDecl[setDefaultConstructor]
                 [addCalls][addCallsToSuper]
end function
% [1-1]change initializer block to method block-----------------------------
rule toMethods
    replace[class_body_declaration]
        Block[block]
    % name begin with initialMethod
    construct MethodID[id]
        initialMethod
    % find a number to add it to the name to be unique
    construct MethodName[id]
        MethodID[!]
    construct MethodCall[declaration_or_statement]
        MethodName();
    import InitCalls [repeat declaration_or_statement]
    % add the new method call to the brevious calls
    export InitCalls
        InitCalls [. MethodCall]
    % lastly the method itself
    construct initialMethod[member_declaration]
        'private 'void MethodName() Block
    by
        initialMethod
end rule
% [1-2]add a method call to constructors-----------------------------------
function addCallsToSuper
    replace*[repeat class_body_declaration]
        Modifiers[repeat modifier]
        ConDeclar[constructor_declarator]
        ThrowsClause[opt throws]
        ConBody[constructor_body]
        Remaining[repeat class_body_declaration]
    %  does the constructor calls its base class?
    where ConBody[containSuper][containThis]
        % yes by a call to this or super
        deconstruct ConBody
        '{
               SuperCallStmt [declaration_or_statement]
               RemainingStmts[repeat declaration_or_statement]
        '}
        % all initialMethods calls
        import InitCalls [repeat declaration_or_statement]
        % join the methods calls within the constructor body
        construct NewBlock[repeat declaration_or_statement]
            InitCalls [. RemainingStmts]
    by
        % the constructor again in the new look
        Modifiers ConDeclar ThrowsClause
        '{
            % a call to base class must be the first stmt
            SuperCallStmt
            NewBlock
        '}
        Remaining[addCallsToSuper]
end function
% [1-3]add a method call to constructors-----------------------------------
function addCalls
    replace*[repeat class_body_declaration]
        Modifiers[repeat modifier]
        ConDeclar[constructor_declarator]
        ThrowsClause[opt throws]
        ConBody[constructor_body]
        Remaining[repeat class_body_declaration]
    %  does the constructor calls its base class?
    where not ConBody[containSuper][containThis]
        % No, there is no a call to this or super
        deconstruct ConBody
        '{
            Stmts[repeat declaration_or_statement]
        '}
        % all initialMethods calls
        import InitCalls [repeat declaration_or_statement]
        % join the methods calls within the constructor body
        construct NewBlock[repeat declaration_or_statement]
            InitCalls [. Stmts]
    by
        % the constructor again in the new look
        Modifiers ConDeclar ThrowsClause
        '{
               NewBlock
        '}
        Remaining[addCalls]
end function
% [1-4] if no constructor is specified create it
function setDefaultConstructor
    replace [repeat class_body_declaration]
        ClassBodyDecl[repeat class_body_declaration]
    % get all constructors
    construct Cons[repeat constructor_declaration]
        _[^ ClassBodyDecl]
    construct ConsCount[number]
        _[length Cons]
    where  ConsCount[< 1]% Not specified
    import ClassName[class_name]
    construct NewConstructor[constructor_declaration]
        ClassName() {}
    by
        NewConstructor
        ClassBodyDecl
end function
% [1-5] if there is at least one initial block
function containInitBlock
    match*[repeat class_body_declaration]
        Block[block]
        remaining[repeat class_body_declaration]
end function
% ------------------------------------------------------------------------
%  ********* [2]STATIC INITIALIZER *********
% ------------------------------------------------------------------------
% [0]find the static initializers ------------------------------------------
function translateStaticInit
    replace[repeat class_body_declaration]
       ClassBodyDecl[repeat class_body_declaration]
    by
        ClassBodyDecl[toStaticConstructor]
end function
% [2-1]change static initializer block to static constructor----------------------------
rule toStaticConstructor
    replace[class_body_declaration]
        'static Block[block]
    % constructor name is same as class name
    import ClassName[class_name]
    construct NewStaticConstructor[constructor_declaration]
       'static ClassName()
          Block
    by
       NewStaticConstructor
end rule
```

A5, TranslateMembers.Rul

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL 103a3
% Rihab Eltayeb, Sudan University, August 2005
% [part of master thesis project]
% CS refers to C#
% % *****
% % common to class body, interface body
% ------------------------------------------------------------------------
% ********* CLASS BODY SECTION *********
% ------------------------------------------------------------------------
% [0]Member Declarations[Constructor,Method,Nested Types(static,nonStatic)----
function translateMemberDeclaration
    replace*[member_declaration]
        Member[member_declaration]
    by
        Member[translateTypeDecl]%2 static nested types only
              %[translateMethodConstructor] 1
end function
% ************************************************************************
% [1]Methods or constructors Declarations-------------------------------
function translateMethodConstructor
    %match only the top level methods or constructors
    skipping [type_declaration]
    replace*[member_declaration]
        MC[method_or_constructor_declaration]
        %deconstruct Member
        %MC[method_or_constructor_declaration]
    by
        MC[translateMethods]
          [doConChaining]% if a super class is called
          [translateConstructors]% modifiers,declarator and block
          [changMethodArray][changeConstructorArray]
end function
% [1-1]Method Declaration-----------------------------------------------
% find a method and further transform its parts
function translateMethods
    replace[method_or_constructor_declaration]
        Modifiers[repeat modifier] TypeSpecify[type_specifier]
        MDeclar[method_declarator]
        ThrowsClause[opt throws]
        MBody[method_body]
    by
        Modifiers[changeMethodModifiers]
        TypeSpecify[changeDataTypes][changeArrayTypes]
        MDeclar[changeMethodDeclarator]
        %ThrowsClause %not allowed in C#
        MBody[translateBlock]
end function
% [1-1-1]---------------------------------------------------
% change the method name and parameters
% If it is the main method then capitalize the letter M
function changeMethodDeclarator
    replace [method_declarator]
       Name[method_name] '( FormalParms[list formal_parameter] ') Dim[repeat
       dimension]
       construct NewFormalParms[list formal_parameter]
        _[changeFormalParamsDataTypes each FormalParms ]

    by
      Name[changeMain] '( NewFormalParms ')Dim

end function
% [1-1-2]---------------------------------------------------
% chang modifiers to C# equivelant and
% preserve the different rules applied for the access
function changeMethodModifiers
    construct JavaModifiers [repeat modifier]
        'native
    construct CSModifiers [repeat modifier]
        'extern
    replace * [ repeat modifier]
        Modifiers [repeat modifier]
    by
    Modifiers [$ each JavaModifiers CSModifiers ]
              [changeAbstract]
              [makeVirtual]
              [changeProtected]
              [removeNonCS]
end function
% [1-1-3]---------------------------------------------------
% abstract members must be public
% if they are not already protected
function changeAbstract
replace [repeat modifier]
       Modifiers[repeat modifier]
    where Modifiers[containAbstract]
    by
       Modifiers[addPublic]
end function
% [1-1-4]---------------------------------------------------
% add virtual if the method is
% not final or abstract or static or private
function makeVirtual
replace [repeat modifier]
        Modifiers[repeat modifier]
    where not Modifiers[containFinal][containAbstract][containStatic][containPrivate]
    by
        'virtual
        Modifiers[addPublic]
end function
% [1-1-5]---------------------------------------------------
% replace protected by two modifiers
% protected and internal
function changeProtected
    replace * [ repeat modifier]
        'protected
    by
        'internal 'protected
end function
% [1-1-6]---------------------------------------------------
% remove final,transient and volatile
% not exist in C#
rule removeNonCS
    replace [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isFinal][isTransient][isVolatile]
    by
       RemainingModifiers
end rule
% [1-1-7]---------------------------------------------------
% virtual or abstract members must be public
% if they are not already private or protected
function addPublic
    replace [repeat modifier]
        Modifiers[repeat modifier]
    where not Modifiers[containPublic][containPrivate][containProtected]
    by
        'public
        Modifiers
end function
% Checking if the modifiers list
% contains specific modifier
% [1-1-8]---------------------------------------------------
rule containFinal
    match [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isFinal]
end rule
% [1-1-9]---------------------------------------------------
rule containAbstract
    match [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isAbstract]
end rule
% [1-1-10]---------------------------------------------------
rule containPublic
    match [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isPublic]
end rule
% [1-1-11]---------------------------------------------------
rule containPrivate
    match [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isPrivate]
end rule
% [1-1-12]---------------------------------------------------
rule containProtected
    match [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isProtected]
end rule
% Checking if a modifier is one of
% final,transient and volatile
% [1-1-13]---------------------------------------------------
function isFinal
    match [modifier]
        'final
end function
% [1-1-14]---------------------------------------------------
function isAbstract
    match [modifier]
        'abstract
end function
% [1-1-15]---------------------------------------------------
function isPublic
    match [modifier]
        'public
end function
% [1-1-16]---------------------------------------------------
function isPrivate
    match [modifier]
        'private
end function
% [1-1-17]---------------------------------------------------
function isProtected
    match [modifier]
        'protected
end function
% [1-1-18]---------------------------------------------------
function isTransient
    match [modifier]
        'transient
end function
% [1-1-19]---------------------------------------------------
function isVolatile
    match [modifier]
        'volatile
end function
% change method name from main to Main
% [1-1-20]---------------------------------------------------
function changeMain
replace[method_name]
    'main
by
    'Main
end function
% [1-1-21]---------------------------------------------------
% change arrays for all formal parameters in the method
function changMethodArray
    replace[method_or_constructor_declaration]
        Modifiers[repeat modifier]
        ReturnedType[type_specifier] Name[declared_name]
        '( FormalParms[list formal_parameter] ') Dim[repeat dimension]
        ThrowsClause[opt throws]
        Body[method_body]
    construct NewFormalParms[list formal_parameter]
        FormalParms[changeMethodConArrayDimentions ]
    by
        Modifiers
        ReturnedType Name
        '( NewFormalParms ') Dim
        ThrowsClause
        Body

end function
% [1-1-22]---------------------------------------------------
% change arrays from int b [] to int []b
rule changeMethodConArrayDimentions
    replace[formal_parameter]
        FormalParm[formal_parameter]
    deconstruct FormalParm
```

```
        FinalOpt[opt 'final] DataType[type_name] Name[declared_name]Dim[repeat
        dimension]
    by
        FinalOpt DataType Dim Name
end rule
% [1-1-23]-----------------------------------------------------
% find a data type within the formal parameter and change
% its data type
function changeFormalParamsDataTypes FormalParm[formal_parameter]
    replace[list formal_parameter]
        FormalParms[list formal_parameter]
    deconstruct FormalParm
        FinalOpt[opt 'final] DataType[type_specifier] Name[variable_name]
    construct NewFormalParm[formal_parameter]
        FinalOpt DataType[changeDataTypes][changeArrayTypes] Name
    by
        FormalParms[, NewFormalParm]
end function
% ****************************************************************************************
% [1-2]Constructor Declaration---------------------------------
% find a constructor and further transform its parts
% a constructor is a special method
function translateConstructors
    replace[method_or_constructor_declaration]
        Modifiers[repeat modifier]
        ConDeclar[constructor_declarator]
        ThrowsClause[opt throws]
        ConBody[constructor_body]

    by
        Modifiers[changeConstructorModifiers]
        ConDeclar[changeConstructorDeclarator]
        %ThrowsClause %not allowed in C#
        ConBody [translateBlock]%[changeConstructorInheritance ConDeclar]

end function
% [1-2-1]-----------------------------------------------------
% change the constructor header
function changeConstructorDeclarator
    replace [constructor_declarator]
       Name[class_name] '( FormalParms[list formal_parameter]') %Init[opt
       constructor_initializer]
       construct NewFormalParms[list formal_parameter]
        _[changeFormalParamsDataTypes each FormalParms ]
    by
       Name '( NewFormalParms')%Init
end function
% [1-2-2]-----------------------------------------------------
% public,protected,private are only allowed
% modifiers for a constructor
function changeConstructorModifiers
    replace * [ repeat modifier]
        Modifiers [repeat modifier]
    by
    Modifiers [setDefaultAccess][changeProtected]
end function
% [1-2-3]-----------------------------------------------------
% if no access modifier is specified
% java default is friendly access,C# default is private access

function setDefaultAccess
    replace [repeat modifier]
        Modifiers [repeat modifier]
        construct ModifiersLength[number]
            _[length Modifiers]
    where
        ModifiersLength[< 1]
    by
        'internal
    end function
% [1-2-4]-----------------------------------------------------
% If the constructor accesses its base class
% that is providing a call to super or this
function doConChaining
    replace[method_or_constructor_declaration]
        Modifiers[repeat modifier]
        ConDeclar[constructor_declarator]
        ThrowsClause[opt throws]
        ConBody[constructor_body]
    %check the first stmt if it is a call to super or this
    where ConBody[containSuper][containThis]
    by
        Modifiers
        ConDeclar[changeToBase ConBody][changeToThis ConBody]
        ThrowsClause %ommited later
        ConBody [removeFirstStmt]
end function
% Checking if the first statemnt in the block
% is super or this
% [1-2-5]-----------------------------------------------------
function containSuper
    match [constructor_body]
       '{
            'super args[repeat component] ';
            Remaining[repeat declaration_or_statement]
       '}
end function
% [1-2-6]-----------------------------------------------------
function containThis
    match [constructor_body]
       '{
            'this args[repeat component] ';
            Remaining[repeat declaration_or_statement]
       '}
end function
% [1-2-7]-----------------------------------------------------
% change super() to : base()
function changeToBase ConBody[constructor_body]
    deconstruct ConBody
    '{
        'super args[repeat component] ';
        Remaining[repeat declaration_or_statement]
    '}
    replace [constructor_declarator]
       Name[class_name] '( FormalParms[list formal_parameter] ')
    by
      Name '( FormalParms ')': 'base args
end function
% [1-2-8]-----------------------------------------------------
% change this() to : this()
function changeToThis ConBody[constructor_body]
    deconstruct ConBody
    '{
        'this args[repeat component] ';
        Remaining[repeat declaration_or_statement]
    '}
    replace [constructor_declarator]
       Name[class_name] '( FormalParms[list formal_parameter] ')
    by
      Name '( FormalParms ')': 'this args
end function
% [1-2-9]-----------------------------------------------------
% after changing remove the super class call
function removeFirstStmt
    replace [constructor_body]
       '{
           stmt[declaration_or_statement]
           Remaining[repeat declaration_or_statement]
       '}
    by
       '{
          Remaining
       '}
end function
% [1-2-10]----------------------------------------------------
% change arrays for all formal parameters in the constructor
function changeConstructorArray
    replace[method_or_constructor_declaration]
        Modifiers[repeat modifier]
        Name[class_name] '( FormalParms[list formal_parameter]')
        ThrowsClause[opt throws]
        Body[constructor_body]
    construct NewFormalParms[list formal_parameter]
        FormalParms[changeMethodConArrayDimentions]
    by
        Modifiers Name '( NewFormalParms ')ThrowsClause
        Body

end function
% ****************************************************************************
% [2]Nested Declarations---------------------------------
% Can have comments as part of declarations
redefine type_declaration
    ...
|   [comment_NL][type_declaration]
end redefine

function translateTypeDecl
    replace[member_declaration]
        NestedType[type_declaration]
    by
        NestedType[nonStaticNested][staticNested]
end function
% [2-1]-----------------------------------------------------
% transform the static nested class
function staticNested
    replace[type_declaration]
        NestedType[type_declaration]
    where not NestedType[isComment]
    where NestedType[checkStatic]
    by
        NestedType[removeStatic][changeStaticClassHeader][changeStaticInterfaceHeader]
              %[removeStatic]
end function
% [2-2]-----------------------------------------------------
% non static or inner classes are not transformed
function nonStaticNested
    replace[type_declaration]
        NestedType[type_declaration]
    where not NestedType[isComment]
    where not NestedType[checkStatic]

    construct S[stringlit]
        "//J2C# NotSupported:Inner classes must be removed manually"
    construct Comment[comment]
        _[unquote S]
    construct RepeatCom[ repeat comment]
        _[. Comment]
    by
        Comment NestedType
end function
% [2-3]-----------------------------------------------------
% fetch for static classes
function checkStatic
    match [type_declaration]
       ClassHead[class_header]ClassBody[class_body]
       deconstruct *[repeat modifier]ClassHead
        Modifiers[repeat modifier]
    where Modifiers[containStatic]
end function
% [2-4]-----------------------------------------------------
% fetch all modifiers
rule containStatic
    match [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isStatic]
end rule
% [2-5]-----------------------------------------------------
% match static key word
function isStatic
    match [modifier]
        'static
end function
% [2-6] match a comment
function isComment
    match [type_declaration]
        Comment[comment]
end function

% [2-7]-----------------------------------------------------
% remove static because it is default in C#
rule removeStatic
    replace [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isStatic]
```

```
    by
        RemainingModifiers
end rule
% [2-8]-------------------------------------------------------
% repeat the whole transformation again
% for the inner static class
function changeStaticClassHeader
    replace [type_declaration]
        ClassHead[class_header]ClassBody[class_body]
        deconstruct ClassHead
            modifiers[repeat modifier] 'class Name[class_name] ExtendClause[opt
            extends_clause] ImplmntClause[opt implements_clause]
        construct NewModifiers [repeat modifier]
            modifiers[changeModifiers ]
        construct NewImplement[opt implements_clause]
            ImplmntClause[changeImplement ExtendClause]
        construct NewExtend [opt extends_clause ]
            ExtendClause[changeExtend]
        construct NewClassHead[class_header]
        NewModifiers 'class Name NewExtend NewImplement
    by
        NewClassHead[addClassExtendToImplmt]
        ClassBody [translateEmptyBody][changeClassBody]
end function
% [2-9]-------------------------------------------------------
% repeat the whole transformation again
% for the inner static interface
function changeStaticInterfaceHeader
    replace [type_declaration]
        InterfaceHead[interface_header] InterfaceBody[interface_body]
        deconstruct InterfaceHead
            modifiers[repeat modifier] 'interface Name[interface_name] ExtendClause[
            opt extends_clause] ImplmntClause[opt implements_clause]
        construct NewModifiers [repeat modifier]
            modifiers[changeModifiers ]
        construct NewImplement[opt implements_clause]
            ImplmntClause[changeImplement ExtendClause]
        construct NewExtend [opt extends_clause]
            ExtendClause[changeExtend]
        construct NewInterfaceHead[interface_header]
        NewModifiers 'interface Name NewExtend NewImplement
    by
        NewInterfaceHead[addInterfaceExtendToImplmt]
        InterfaceBody [translateEmptyBody][changeInterfaceBody]
end function

% ------------------------------------------------------------------------
% ********* INTERFACE BODY SECTION  *********
% ------------------------------------------------------------------------
% [1]Method Declaration---------------------------------------------------
% find a method and further transform its parts
function translateIntMethods
    replace[repeat class_body_declaration]
        Modifiers[repeat modifier] TypeSpecify[type_specifier]
        MDeclar[method_declarator]
        ThrowsClause[opt throws]
        MBody[method_body]
        RemainingRepeatBodyDecl[repeat class_body_declaration]
```



```
    by
        Modifiers[changeIntMethodModifiers]
        TypeSpecify[changeDataTypes][changeArrayTypes]
        MDeclar[changeMethodDeclarator]
        %ThrowsClause %not allowed in C#
        MBody[translateBlock]
        RemainingRepeatBodyDecl[translateIntMethods]
end function
% [2]---------------------------------------------------------
% chang modifiers to C# equivelant and
% preserve the different rules applied for the access
function changeIntMethodModifiers
    construct JavaModifiers [repeat modifier]
        'native
    construct CSModifiers [repeat modifier]
        'extern
    replace * [ repeat modifier]
        Modifiers [repeat modifier]
    by
        Modifiers [$ each JavaModifiers CSModifiers ]
            [removeIntPublic]% C# default is public
            %[changeAbstract][makeVirtual]not needed
            %[changeProtected][removeNonCS]not needed
end function
% [3]---------------------------------------------------------
% remove public because it is default in C#
rule removeIntPublic
    replace [repeat modifier]
       CurrentModifier[modifier]
       RemainingModifiers[repeat modifier]
    where CurrentModifier[isPublic]
    by
       RemainingModifiers
end rule
```

A6, TranslateFieldDeclaration.Rul

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL 103a3
% Rihab Eltayeb, Sudan University, August 2005
% [part of master thesis project]
% % *****
% % common to field declarations in class/interface
% Note: not applied to multiple declarations in one line
% ------------------------------------------------------------------------
%  ********* [1]CLASS  FIELD DECLARATIONS    *********
% ------------------------------------------------------------------------
% [0]find a field declaration in a class body
function translateFieldDeclaration
    replace[repeat class_body_declaration]
        ClassBodyDecl[repeat class_body_declaration]
    by
        ClassBodyDecl[changeField]
end function
% [1-1]apply other functions to a field declaration----------------------------
function changeField
    replace*[repeat class_body_declaration]
        FieldDecl[field_declaration]
        Remaining[repeat class_body_declaration]
    by
       FieldDecl[changeFieldArrayDimensions][checkVar][checkPrimitiveConstants]
            [checkRunTimeConstants]% [changeArrayDimensions]
       Remaining[changeField]
end function
% [1-2]for constants that has predefined value in their declaration------------
function checkPrimitiveConstants
    replace[field_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_declarators];
    construct AllVarDecl[repeat variable_declarator]
        _[^ VarDecl]
    deconstruct AllVarDecl
        FirstVarDecl[variable_declarator]Remaining[repeat variable_declarator]
    where Modifiers[containFinal]
    where FirstVarDecl[isCompileTime]
    by
        Modifiers[removeStatic][finalToConst]
             [removeNonCSModifiers][changeProtected]
        TypeSpecifier[changeDataTypes][changeArrayTypes]
        VarDecl ;
end function
% [1-3]for constants that has no immediate value in their declaration----------
function checkRunTimeConstants
    replace[field_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_declarator];
    where Modifiers[containFinal]
    where not VarDecl[isCompileTime]
    by
        Modifiers[finalToReadOnly][removeNonCSModifiers][changeProtected]
        TypeSpecifier[changeDataTypes][changeArrayTypes]
        VarDecl ;
end function
% [1-4]for field declaration as normal variable declaration--------------------
function checkVar
    replace[field_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_declarators];
    construct AllVarDecl[repeat variable_declarator]
        _[^ VarDecl]
    deconstruct AllVarDecl
        FirstVarDecl[variable_declarator]Remaining[repeat variable_declarator]
    where not Modifiers[containFinal]
    by
        Modifiers[removeNonCSModifiers][changeProtected]
        TypeSpecifier[changeDataTypes][changeArrayTypes]
        VarDecl ;
end function
% [1-5]change a field declaration int b[] to int[]b ---------------------------
function changeFieldArrayDimensions
    replace[field_declaration]
        Modifiers[repeat modifier]
        DataType[type_name]
        VarName[declared_name]Dim [repeat dimension]VarInit [opt
        equals_variable_initializer];
    % data type first and [] follows
    construct NewTypeSpecifier[type_specifier]
        DataType Dim
    by
        Modifiers NewTypeSpecifier VarName VarInit ;
end function
% [1-6]check if it has a number,true,false,string or char literal-----------------
function isCompileTime
    match[variable_declarator]
        VarName[variable_name] '= Value[literal]
end function
% [1-7]change the final key word to const-------------------------------------
function finalToConst
    replace *[repeat modifier]
        CurrentModifier[modifier]
        RemainingModifiers[repeat modifier]
    where CurrentModifier[isFinal]
    by
        'const
        RemainingModifiers
end function
% [1-8]change the final key word to readonly----------------------------------
function finalToReadOnly
    replace *[repeat modifier]
        CurrentModifier[modifier]
        RemainingModifiers[repeat modifier]
    where CurrentModifier[isFinal]
    by
        'readonly
        RemainingModifiers
end function
% [1-9] remove transient and volatile
%  not exist in C#
rule removeNonCSModifiers
    replace [repeat modifier]
        CurrentModifier[modifier]
        RemainingModifiers[repeat modifier]
    where CurrentModifier[isTransient][isVolatile]
    by
        RemainingModifiers
end rule
% ------------------------------------------------------------------------
%  ********* [2]INTERFACE  FIELD DECLARATIONS    *********
% ------------------------------------------------------------------------
% [0]find a field declaration in an interface body
function translateIntFieldDeclaration
    replace[repeat class_body_declaration]
        ClassBodyDecl[repeat class_body_declaration]
    by
        ClassBodyDecl[changeIntField]
end function
% [2-1]apply other functions to a field declaration----------------------------
function changeIntField
    replace*[repeat class_body_declaration]
        FieldDecl[field_declaration]
        Remaining[repeat class_body_declaration]
    construct S[stringlit]
        "//J2C# Unsupported:Interface field must be removed manually"
    construct Comment[comment]
        _[unquote S]
    by
        Comment
        FieldDecl
        Remaining[changeIntField]
end function
```

A7, TranslateBlockStatements.Rul

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL 103a3
% Rihab Eltayeb, Sudan University, August 2005
% [part of master thesis project]
% % *****
% % common to Statements and control structures
%
% ---------------------------------------------------------------------
% *********  Block STATEMENTS  *********
% ---------------------------------------------------------------------
% [0]find a block as the body of a method or constructor or a statementt
function translateBlock
    replace*[block]
       '{
            DeclOrStmt[repeat declaration_or_statement]
        '}
    by
       '{
        DeclOrStmt[translateVarDeclaration]
                  [translateClassInBlock]
                      [translateStatementInBlock]
        '}
end function
% ***********************************************************************
% [1-1]VARIABLE DECLARATIONS-----------------------------------------
function translateVarDeclaration
replace*[repeat declaration_or_statement]
    Var[local_variable_declaration]
    Remaining[repeat declaration_or_statement]
by
    Var[checkLocalVars]
       [checkLocalConstants][checkLocalRunTimeConstants]
       [checkLocalBlankConstants][changeArrayDimensions]
    Remaining[translateVarDeclaration]
end function
% [1-1-1]for constants that has predefined value in their declaration------------
function checkLocalConstants
    replace[local_variable_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_declarators];
    construct AllVarDecl[repeat variable_declarator]
        _[^ VarDecl]
    deconstruct AllVarDecl
        FirstVarDecl[variable_declarator]Remaining[repeat variable_declarator]
    where Modifiers[containFinal]
    where FirstVarDecl[isCompileTime]
    by
        Modifiers[finalToConst][changeProtected]
        TypeSpecifier[changeDataTypes][changeArrayTypes][changeExceptionName]
        VarDecl[changeCSException][changeCSStatement] ;
end function
% [1-1-2]for constants that has no immediate value in their declaration----------
% final is not permitted in C# so change it to a variable
% comment must be added
function checkLocalRunTimeConstants
    replace[local_variable_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_declarators];
    construct AllVarDecl[repeat variable_declarator]
        _[^ VarDecl]
    deconstruct AllVarDecl
        FirstVarDecl[variable_declarator]Remaining[repeat variable_declarator]
    where Modifiers[containFinal]
    where not FirstVarDecl[isCompileTime]
    by
        Modifiers[changeProtected][removeNonCS]%remove final
        TypeSpecifier[changeDataTypes][changeArrayTypes][changeExceptionName]
        VarDecl[changeCSException][changeCSStatement] ;
end function
% [1-1-3]for blank constants with no initializer
function checkLocalBlankConstants
    replace[local_variable_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_name];%no initializer
    where Modifiers[containFinal]
    by
        Modifiers[changeProtected][removeNonCS]%remove final
        TypeSpecifier[changeDataTypes][changeArrayTypes][changeExceptionName]
        VarDecl[changeCSException][changeCSStatement] ;
end function
% [1-1-4]for normal local variable declaration
function checkLocalVars
    replace*[local_variable_declaration]
        Modifiers[repeat modifier]
        TypeSpecifier[type_specifier]
        VarDecl[variable_declarators];
        construct AllVarDecl[repeat variable_declarator]
        _[^ VarDecl]
    deconstruct AllVarDecl
        FirstVarDecl[variable_declarator]Remaining[repeat variable_declarator]
    where not Modifiers[containFinal]
    by
        Modifiers[removeNonCSModifiers][changeProtected]
        TypeSpecifier[changeDataTypes][changeArrayTypes][changeExceptionName]
        VarDecl[changeCSException][changeCSStatement] ;
end function
% ***********************************************************************
% [1-2]ARRAY DECLARATIONS---------------------------------------------
% change int b[] to int[]b
function changeArrayDimensions
    replace[local_variable_declaration]
        Modifiers[repeat modifier]
        DataType[type_name]
        VarName[declared_name]Dim [repeat dimension]VarInit [opt
        equals_variable_initializer];
    construct NewTypeSpecifier[type_specifier]
        DataType Dim
    by
        Modifiers NewTypeSpecifier VarName VarInit ;
end function
% ***********************************************************************
% [2]CLASS DECLARATION-----------------------------------------------
% no transformation just comment for this.
function translateClassInBlock
    replace*[repeat declaration_or_statement]
        ClassInBlock[class_declaration]
        Remaining[repeat declaration_or_statement]
        construct S[stringlit]
        "//J2C# Unsupported:Classes within blocks must be removed manually"
        construct Comment[comment]
            _[unquote S]
    by
        Comment
        ClassInBlock
        Remaining[translateVarDeclaration]
end function
% ***********************************************************************
% [3]STATEMENTS DECLARATIONS-----------------------------------------
function translateStatementInBlock
    replace*[repeat declaration_or_statement]
        Stmnt[statement]
        Remaining[repeat declaration_or_statement]
    by
        Stmnt[translateStatement]
        Remaining[translateStatementInBlock]
end function
% [3-1]one statement at a time---------------------------------------
function translateStatement
    replace [statement]
        OneStmnt[statement]
    by
        OneStmnt[changeExpressionStmt]
                [changeIf]
                [changeSwitch]
                [changeWhile]
                [changeDoWhile]
                [changeFor]
                [changeBreak]
                [changeContinue]
                [changeReturn]
                [changeThrow]
                [changeSynchronized]
                [changeTry]

end function
% [3-2]Expression Statement------------------------------------------
function changeExpressionStmt
    replace[statement]
        Expression[expression];
    by
        Expression[changeExpression][changeCSStatement];
end function
function changeExpression
    replace*[expression]
        Expression[expression]
    by
        Expression[$ '>>> '>>]
end function

% [3-3]If Statement--------------------------------------------------
function changeIf
    replace*[statement]
        'if '( IfExp[expression] ')
                IfStmt[statement]
            ElseClause[opt else_clause]
    by
        'if '( IfExp[changeExpression] ')
            IfStmt[translateStatement][translateBlock]
            ElseClause[changeElseClause]
end function
% [3-4]else clause---------------------------------------------------
function changeElseClause
    replace[opt else_clause]
        'else ElseStmts [statement]
    by
        'else ElseStmts[translateStatement][translateBlock]
end function
% [3-5]Switch Statement----------------------------------------------
function changeSwitch
replace*[statement]
    'switch '( SwitchExp[expression] ') '{ SwitchAlters [repeat switch_alternative] '}
by
    'switch '( SwitchExp[changeExpression] ')'{ SwitchAlters [addBreak][fallThrough]
                                        [changeSwitchStmts] '}
end function
% [3-5-1]In C# the last case alternative has to have a break
function addBreak
    replace[repeat switch_alternative]
        SwitchAlters [repeat switch_alternative]
    construct Length[number]
         _[length SwitchAlters]
    construct Index[number]
        Length[- 1]
    construct LastAlter[repeat switch_alternative]
        SwitchAlters [tail Length]
    deconstruct LastAlter
        Label[switch_label]  Stmts[repeat declaration_or_statement]
    %no break ?
    where not Stmts[ContainBreak]
    % all alternatives before the last one
    construct BeforeLastAlter[repeat switch_alternative]
        SwitchAlters [head Index]
    %a new break to be added
    construct Break[repeat declaration_or_statement]
        break ;
    construct NewStmts[repeat declaration_or_statement]
        Stmts[. Break]
    %new statements with break
    construct NewLastAlter[repeat switch_alternative]
        Label NewStmts
    construct NewSwitchAlters[repeat switch_alternative]
        BeforeLastAlter[. NewLastAlter]
    by
        NewSwitchAlters
end function
% [3-5-2]check if the last statements contain a break
function ContainBreak
    match[repeat declaration_or_statement]
        Stmts[repeat declaration_or_statement]
    construct Length[number]
         _[length Stmts]
    construct LastDecStmt[repeat declaration_or_statement]
        Stmts [tail Length]
    deconstruct *LastDecStmt
```

```
        breakStmt[break_statement]
end function
% [3-5-3]check if the last statements contain a goto
function isGoto
match[repeat declaration_or_statement]
        Stmts[repeat declaration_or_statement]
    construct Length[number]
         _[length Stmts]
    construct LastDecStmt[repeat declaration_or_statement]
        Stmts [tail Length]
    deconstruct *LastDecStmt
        gotoStmt[goto_statement]
end function
% [3-5-4]C# does not allow fall through( case without break or goto)
% unless the case label contains no statements.
function fallThrough
    replace*[repeat switch_alternative]
        Label[switch_label]  Stmts[repeat declaration_or_statement]
        RemainingSwitchAlters [repeat switch_alternative]
    construct Length[number]
         _[length Stmts]
    %no break and at least one statement is present
    where not Stmts[ContainBreak]
    where Length[ > 0]
    where not Stmts[isGoto]
    deconstruct RemainingSwitchAlters
        NextLabel[switch_label]  NextStmts[repeat declaration_or_statement]
        RemainingAlters [repeat switch_alternative]
    %goto the next alternative
    construct GotoStmt[repeat declaration_or_statement]
        goto NextLabel
    %replace colon with semi colon to be a valid statement
    construct NewGotoStmt[repeat declaration_or_statement]
        GotoStmt[$ ': '; ]
    construct NewStmts[repeat declaration_or_statement]
        Stmts[. NewGotoStmt]
    %new statements with goto
    construct NewAlter[repeat switch_alternative]
        Label NewStmts
    %check the other cases also
    construct NewRemainingAlters[repeat switch_alternative]
        RemainingSwitchAlters[fallThrough]
    %the final result
    construct NewSwitchAlters[repeat switch_alternative]
        NewAlter[. NewRemainingAlters]
    by
        NewSwitchAlters
end function
% [3-5-5] to translate the statements inside the switch case
function changeSwitchStmts
replace*[repeat switch_alternative]
        Label[switch_label]  Stmts[repeat declaration_or_statement]
        RemainingSwitchAlters [repeat switch_alternative]
    by
        Label  Stmts[translateStatementInBlock]
        RemainingSwitchAlters[changeSwitchStmts]

end function
```

```
% [3-6]While Statement-------------------------------------------------------
function changeWhile
    replace*[statement]
        'while '( WhileExp[expression] ')
                Stmts[statement]
    by
        'while '( WhileExp[changeExpression]')
                Stmts[translateStatement][translateBlock]
end function

% [3-7]While Statement-------------------------------------------------------
function changeDoWhile
    replace*[statement]
        'do
            Stmts[statement]
        'while '( DoWhileExp[expression] ');
    by
        'do
            Stmts[translateStatement][translateBlock]
        'while '( DoWhileExp[changeExpression]');
end function
% [3-8]For Statement---------------------------------------------------------
function changeFor
    replace*[statement]
        'for '( InitExp[for_init] ForExp[for_expression]UpdateExp [for_update] ')
                Stmts[statement]
    by
        'for '( InitExp[checkLocalVars][changeExpression] ForExp[changeExpression]
        UpdateExp [changeExpression] ')
                Stmts[translateStatement][translateBlock]
end function
% [3-9]Break Statement-------------------------------------------------------
function changeBreak
    replace*[statement]
        BreakStmt[break_statement]
    deconstruct BreakStmt
        'break LabelName[reference]';
    by
        BreakStmt[$ 'break 'goto]%'goto LabelName ';
end function
% [3-10]Continue Statement---------------------------------------------------
function changeContinue
    replace*[statement]
        ContinueStmt[continue_statement]
    deconstruct ContinueStmt
        'continue LabelName[reference]';
    by
        ContinueStmt[$ 'continue 'goto]%'goto LabelName ';
end function
% [3-11]Return Statement-----------------------------------------------------
function changeReturn
    replace*[statement]
        'return ReturnExp[opt expression] ';
    by
        'return ReturnExp[changeOptExpression] ';
end function
% [3-12]Throw Statement------------------------------------------------------
% The Java exception name [see table ] is matched and converted to the C# equivalent one.
function changeThrow
```

```
    replace*[statement]
        'throw ThrowExp[expression] ';
    by
        'throw ThrowExp[changeExpression][changeCSException][changeCSStatement] ';
end function
% [3-13]Synchronized Statement-----------------------------------------------
function changeSynchronized
    replace*[statement]
            'synchronized '( SyncExp[expression] ')
            Stmts[statement]
        by
            'lock '( SyncExp[changeExpression] ')
            Stmts[translateStatement][translateBlock]
end function
% [3-14]Try Statement--------------------------------------------------------
function changeTry
    replace*[statement]
        'try TryBlock[block] Catches[repeat catch_clause] Finally[opt finally_clause]
    construct NewCatches[repeat catch_clause]
        _[changeCatch each Catches]
    by
        'try TryBlock[translateBlock] NewCatches Finally[changeFinally]
end function
% %[3-14-1]find a catch clause and change its data type-------------------------
function changeCatch CatchClause[catch_clause]
    replace[repeat catch_clause]
        Catches[repeat catch_clause]
    deconstruct CatchClause
        'catch '( DataType[type_specifier] Name[variable_name] ')
            CatchBlock[block]
    construct NewCatchClause[catch_clause]
        'catch '( DataType[changeDataTypes][changeArrayTypes][changeExceptionName]
        Name ')
            CatchBlock[translateBlock]
    by
        Catches[. NewCatchClause]
end function
% [3-14-2]find a finally clause and change its block-----------------------------
function changeFinally
    replace[opt finally_clause]
        'finally FinBlock[block]
    by
        'finally FinBlock[translateBlock]
end function
% [4]*****************************************API Part
% to fetch for a call for a method in a class
% mainly used to change System classes such as out.println or print methods
% [4-1]this fuction should be changed to satisfy other patterns of method calls
function changeCSStatement
    replace [expression]
        FirstId[id] '. SecondId[id]'. ThirdId[id]Remaining[repeat component]
    %to search for specified entry in the statemnts table
    construct Key[reference]
        FirstId.SecondId.ThirdId
    import StatementMapping [StmtMapper]
    % match the key entry with C# one
    deconstruct *[table_entry] StatementMapping
        Key -> CSStmt [reference]
    % to build a new valid expression
```

```
    deconstruct CSStmt
        FirstCSId[id] CSRemaining[repeat component]
    %join the method arguements with the new part
    by
        FirstCSId CSRemaining[. Remaining]
end function
% *****************************************API Part
% [4-2]for changing the exception names
function changeCSException
    replace *[expression]
        exp[expression]
    deconstruct exp
        'new ExceptionName[qualified_name]()
    by
        'new ExceptionName[changeExceptionName]()
end function
% Generally for optional expression
function changeOptExpression
    replace[opt expression]
        Expression[opt expression]
    by
        Expression[$ '>>> '>>]
end function
```

A8, HelperRules.Rul

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL 103a3 Translation, Java to C#
% Rihab Eltayeb, Sudan University, August 2005
% [part of master thesis project]

% ---------------------------------------------------------------------------
% *********  DATA TYPES SECTION    *********
% ---------------------------------------------------------------------------
% general rules used in more than one transformation

% [1] first find a java primitive
% then change it
function changeDataTypes
    replace[type_specifier]
        DataType[primitive_type]
    by
        DataType[changePrimDataTypes]
end function
% [2]To change the primitive data types
% byte to sbyte,boolean to bool
% other data types are the same
function changePrimDataTypes
    replace [primitive_type]
        JavaType[primitive_type]
        import PrimDataTypesMapping [DataMapper]
        deconstruct * [primitiveTable] PrimDataTypesMapping
            JavaType -> CSType [primitive_type]
    by
        CSType
end function
% [3]changing the array declaration if it secifies
% a primitive data type as its elements type
function changeArrayTypes
    replace[type_specifier]
        ArrayType[type_name] Dimentions[repeat dimension+]
        deconstruct *[primitive_type] ArrayType
            PrimType[primitive_type]
    by
        PrimType[changePrimDataTypes]Dimentions
end function
% [4]change the exception name with the C# one
% Exception names can be found in throw statement or
% in variable declaration or initializer
function changeExceptionName
replace *[qualified_name]
    ExceptionName[qualified_name]
import RunTimeExceptionsMapper [ExceptionMapper]
deconstruct * [exceptionTable] RunTimeExceptionsMapper
    ExceptionName -> CSName [qualified_name]
by
    CSName
end function
```

A9, JavaToC#.Txl

Die folgenden Seiten zeigen den Quellcode der Datei.

```
% TXL Java 1.1 Basis Grammar
% Copyright 2001 TXL Software Research Inc. and James R. Cordy

% Adapted from the Java Language Specification, 2nd edition

% Modified by Xinping Guo May 28, 2002
% -- Defined a new nonterminal type called "declared_name".
% -- "class_name", "interface_name", "variable_name" and "method_name" are "declared_name".
% -- Changed "reference" to "variable_name" in "catch_clause".
% -- In "constructor_declarator", changed "type_name" to "class_name".
% -- In "implements_clause", changed "class_name+" to "qualified_name+".
% -- Defined a new nonterminal type called "method_or_constructor_declaration".
% -- In "member_declaration", removed "field_declaration" and changed
%        "method_declaration" to "method_or_constructor_declaration".
% -- In "class_body_declaration", removed "constructor_declaration" and
%        added "field_declaration".

% Modified by Xinping Guo June 12, 2002
% The changes made expansion of dot notation much easier.
% -- Redefined "primary". Removed "primary_component" and "dot_primary_component"
% -- Redefined "subprimary".
% -- Removed "field_access", "method_invocation" and "constructor_invocation".
% -- Redefined "reference".
% -- Defined "component".
% -- Defined "method_argument".

% Modified by Xinping Guo July 17, 2002
% -- Added [repeat component] after "( [expression] ') in defintion of primary.

% Modified by Xinping Guo July 23, 2002
% -- Added 'strictfp in defintion of modifier.



% Lexical conventions of Java

#pragma -width 128

tokens
    number    "\d+[LlFfDd]?"                                    % Java ints
              + "[(\d+(.\d*))(.\d+)]([eE][+-]?\d+)?[LlFfDd]?"   % Java float constants
              + "0[Xx][\dABCDEFabcdef]+[LlFfDd]?"               % Java hex constants
end tokens

comments
    //
    /*  */
end comments

compounds
    !=  '% =  &&  &=  *=  ++  +=   --  -=  /=  ^=  |=  ||
    <<  <<=  <=  ==  >=  >>  >>=  >>>  >>>=
end compounds


% Removed this and super - XG June 19, 2002
keys
    'abstract 'boolean 'break 'byte 'case 'cast 'catch 'char 'class 'const 'continue
    'default 'do 'double 'else 'extends 'false 'final 'finally 'float 'for 'future
    'goto 'if 'implements 'import 'inner 'instanceof 'int 'interface 'long
    'native 'new 'null 'package 'private 'protected 'public 'return
    'short 'static  'switch 'synchronized
    'throw 'throws 'transient 'true 'try 'void 'volatile 'while
end keys


% Syntax of Java 1.1

define program
    [package_declaration]
end define


% Declarations

define package_declaration
    [opt package_header]
    [repeat import_declaration]
    [repeat type_declaration]
end define

define package_header
    'package [package_name] '; [NL][NL]
end define

define package_name
    [qualified_name]
end define

define import_declaration
    'import [imported_name] '; [NL][NL]
end define

define imported_name
    [package_or_type_name] [opt dot_star]
end define

define package_or_type_name
    [qualified_name]
end define

define dot_star
    '. '*
end define

define declared_name
   [reference]
end define

% Class and interface declarations

define type_declaration
        [class_declaration]      [NL][NL]
    |   [interface_declaration]  [NL][NL]
end define

define class_declaration
    [class_header] [class_body]
end define

define class_header
    [repeat modifier] 'class [class_name] [opt extends_clause] [opt implements_clause]
end define

define class_body
    [class_or_interface_body]
end define

define interface_declaration
    [interface_header] [interface_body]
end define

define interface_header
    [repeat modifier] 'interface [interface_name] [opt extends_clause] [opt
    implements_clause]
end define

define interface_body
    [class_or_interface_body]
end define

define modifier
        'abstract
    |   'final
    |   'public
    |   'protected
    |   'private
    |   'static
    |   'transient
    |   'volatile
    |   'native
    |   'synchronized
    |   'strictfp       % Added strictfp XG July 23, 2002
end define

define extends_clause
    'extends [list type_name+]
end define

define implements_clause
    'implements [list qualified_name+]
end define

define class_name
    [declared_name]
end define

define interface_name
    [declared_name]
end define

define class_or_interface_body
    '{                                          [NL][IN]
       [repeat class_body_declaration]          [EX]
    '} [opt ';]                                 [NL][NL]
end define

define class_body_declaration
        [empty_declaration]
    |   [member_declaration]
    |   [instance_initializer]
    |   [static_initializer]
    |   [field_declaration]
end define

define empty_declaration
    '; [NL]
    |'//
end define

define member_declaration
       [method_or_constructor_declaration]
    |  [type_declaration]     % [class_declaration] or [interface_declaration]
end define

define method_or_constructor_declaration
        [method_declaration]
    |   [constructor_declaration]
end define

define instance_initializer
    [NL] [block]              [NL][NL]
end define

define static_initializer
    [NL] 'static [block]      [NL][NL]
end define

define constructor_declaration
    [NL] [repeat modifier] [constructor_declarator] [opt throws] [constructor_body] [
    NL][NL]
end define

define constructor_declarator
    [class_name] '( [list formal_parameter] ')
end define

define constructor_body
    [block] [NL]
end define


% Field declarations

define field_declaration
    [variable_declaration]
end define

define variable_declaration
    [repeat modifier] [type_specifier] [variable_declarators] '; [NL]
end define

define variable_declarators
    [list variable_declarator+]
end define

define variable_declarator
```

```
    [variable_name] [opt equals_variable_initializer]
end define

define variable_name
    [declared_name][repeat dimension]
end define

define equals_variable_initializer
    '= [variable_initializer]
end define

define variable_initializer
        [expression]
    |   [array_initializer]
end define

define array_initializer
    '{ [list variable_initializer] [opt ',] '}
end define


% Method declarations

define method_declaration
    [NL] [repeat modifier] [type_specifier] [method_declarator] [opt throws] [
    method_body]
end define

define method_declarator
    [method_name] '( [list formal_parameter] ') [repeat dimension]
end define

define method_name
    [declared_name]
end define

define formal_parameter
    [opt 'final] [type_specifier] [formal_parameter_name]
end define

define formal_parameter_name
    [variable_name]
end define

define throws
    'throws [list qualified_name+]
end define

define method_body
        [block]         [NL][NL]
    |   ';              [NL][NL]
end define


% Type specifiers

define type_specifier
        [type_name]
    |    [array_type_specifier]


end define

define array_type_specifier
    [type_name] [repeat dimension+]
end define

define type_name
        [primitive_type]
    |   [qualified_name]
end define

define primitive_type
        'boolean
    |   'char
    |   'byte
    |   'short
    |   'int
    |   'long
    |   'float
    |   'double
    |   'void
end define


% Statements

define block
    '{                                          [NL][IN]
        [repeat declaration_or_statement]       [EX]
    '}
end define

define declaration_or_statement
        [local_variable_declaration]
    |   [class_declaration]
    |   [statement]
end define

define local_variable_declaration
    [variable_declaration]
end define

define statement
        [label_statement]
    |   [empty_statement]
    |   [expression_statement]
    |   [if_statement]
    |   [switch_statement]
    |   [while_statement]
    |   [do_statement]
    |   [for_statement]
    |   [break_statement]
    |   [continue_statement]
    |   [return_statement]
    |   [throw_statement]
    |   [synchronized_statement]
    |   [try_statement]
    |   [block]
end define


define empty_statement
    ';                      [NL]
end define

define label_statement
    [label_name] ':         [NL]
end define

define label_name
    [reference]
end define

define expression_statement
    [expression] ';         [NL]
end define

define if_statement
    'if '( [expression] ')
        [statement]
    [opt else_clause]       [NL]
end define

define else_clause
    'else
        [statement]
end define

define switch_statement
    'switch '( [expression] ') [switch_block]  [NL]
end define

define switch_block
    '{                                          [IN][NL]
        [repeat switch_alternative]             [EX]
    '}
end define

define switch_alternative
    [switch_label]                              [IN][NL]
        [repeat declaration_or_statement]       [EX]
end define

define switch_label
        'case [constant_expression] ':
    |   'default ':
end define

define while_statement
    'while '( [expression] ')
        [statement]                             [NL]
end define

define do_statement
    'do
        [statement]
    'while '( [expression] ') ';                [NL]
end define


define for_statement
    'for '( [for_init] [for_expression] [for_update] ')
        [statement]                             [NL]
end define

define for_init
        [list expression] ';
    |   [local_variable_declaration]
end define

define for_expression
    [opt expression] ';
end define

define for_update
    [list expression]
end define

define break_statement
    'break [opt label_name] ';        [NL]
end define

define continue_statement
    'continue [opt label_name] ';     [NL]
end define

define return_statement
    'return [opt expression] ';       [NL]
end define

define throw_statement
    'throw [expression] ';            [NL]
end define

define synchronized_statement
    'synchronized '( [expression] ')
        [statement]                   [NL]
end define

define try_statement
    'try [block] [repeat catch_clause] [opt finally_clause]     [NL]
end define

define catch_clause
%   'catch '( [type_specifier] [reference] ') [block]
    'catch '( [type_specifier] [variable_name] ') [block]  % July 15
end define

define finally_clause
    'finally [block]
end define


% Expressions

define constant_expression
    [expression]    % which is a constant
end define
```

```
define expression
    [assignment_expression]
end define

define assignment_expression
        [conditional_expression]
    |   [unary_expression] [assignment_operator] [assignment_expression]
end define

define assignment_operator
        '=
    |   '*=
    |   '/=
    |   '% =
    |   '+=
    |   '-=
    |   '<<=
    |   '>>=
    |   '>>>=
    |   '&=
    |   '^=
    |   '|=
end define

define conditional_expression
    [conditional_or_expression] [opt conditional_choice]
end define

define conditional_choice
    '? [expression] ': [conditional_expression]
end define

define conditional_or_expression
    [conditional_and_expression] [repeat or_conditional_and_expression]
end define

define or_conditional_and_expression
    '|| [conditional_and_expression]
end define

define conditional_and_expression
    [inclusive_or_expression] [repeat and_inclusive_or_expression]
end define

define and_inclusive_or_expression
    '&& [inclusive_or_expression]
end define

define inclusive_or_expression
    [exclusive_or_expression] [repeat or_exclusive_or_expression]
end define

define or_exclusive_or_expression
    '| [exclusive_or_expression]
end define

define exclusive_or_expression
    [and_expression] [repeat or_and_expression]
end define

define or_and_expression
    '^ [and_expression]
end define

define and_expression
    [equality_expression] [repeat and_equality_expression]
end define

define and_equality_expression
    '& [equality_expression]
end define

define equality_expression
    [relational_expression] [repeat equality_op_relational_expression]
end define

define equality_op_relational_expression
    [equality_op] [relational_expression]
end define

define equality_op
    '== | '!=
end define

define relational_expression
    [shift_expression] [repeat relational_op_shift_expression]
end define

define relational_op_shift_expression
        [relational_op] [shift_expression]
    |   'instanceof [type_specifier]
end define

define relational_op
    '<  | '> | '<= | '>=
end define

define shift_expression
    [additive_expression] [repeat shift_additive_expression]
end define

define shift_additive_expression
    [shift_op] [additive_expression]
end define

define shift_op
    '<< | '>> | '>>>
end define

define additive_expression
    [multiplicative_expression] [repeat add_op_multiplicative_expression]
end define

define add_op_multiplicative_expression
    [add_op] [multiplicative_expression]
end define

define add_op
    '+ | '-
end define

define multiplicative_expression
    [unary_expression] [repeat mult_op_unary_expression]
end define

define mult_op_unary_expression
    [mult_op] [unary_expression]
end define

define mult_op
    '* | '/ | '%
end define

define unary_expression
        [pre_inc_dec] [unary_expression]
    |   [unary_op] [unary_expression]
    |   [postfix_expression]
    |   [cast_expression]
end define

define pre_inc_dec
    '++ | '--
end define

define unary_op
    '+ | '- | '~ | '!
end define

define cast_expression
    '( [type_specifier] ') [unary_expression]
end define

define postfix_expression
    [primary] [repeat post_inc_dec]
end define

define post_inc_dec
    '++ | '--
end define

define subscript
    '[ [expression] ']
end define

define primary
        [literal]
    |   [reference]
    |   '( [expression] ') [repeat component]
    |   [class_instance_creation_expression]
    |   [array_creation_expression]
end define

define literal
        [numeric_literal]
    |   [character_literal]
    |   [string_literal]
    |   [boolean_literal]
    |   [null_literal]
end define

define numeric_literal
    [number]
end define

define character_literal
    [charlit]
end define

define string_literal
    [stringlit]
end define

define boolean_literal
        'true
    |   'false
end define

define null_literal
    'null
end define

define class_instance_creation_expression
    'new [class_or_interface_name] '( [list argument] ') [opt class_body]
end define

define class_or_interface_name
    [qualified_name]
end define

define array_creation_expression
    'new [array_type_specifier] [opt array_initializer]
end define

define dimension
    '[ [opt expression] ']
end define

define argument
    [expression]
end define

define qualified_name
    [reference]
end define

define reference
    [id] [repeat component]
end define

define component
      [dot_id]
   | [method_argument]
   | [subscript]
end define

define method_argument
```

```
    '( [list argument] ')
end define

define dot_id
    '. [id]
end define

% --- END ---

```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Chemnitz, den 25.05..2014

Robert Morgenstern